Redirect:

- The Flask class has a redirect() function. When called, it returns a response object and redirects the user to another target location with specified status code.
- General syntax: **redirect(location, statuscode, response)**
  The location parameter is the URL where response should be redirected.
  The statuscode parameter is sent to the browser's header. The default value is 302.
  The response parameter is used to instantiate response.
  Note: Of the 3 parameters above, only the location parameter is required.
- As shown below, the redirect function is very useful when combined with the url_for function.

URL_FOR:

- The **url_for()** function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument, and one or more keyword arguments, each corresponding to the variable part of URL.
- General syntax: **url_for(function_name)**
- **Note:** url_for() can be used with redirect() to redirect the user to another function and another webpage.
- **Note:** We can also use url_for in our HTML code.

Sessions:

- **Session** is the time interval when a client logs into a server and logs out of it. The data, which is needed to be held across this session, is stored in the client's browser.
  Note: Suppose the client logs into Gmail using Google Chrome. Because the session data is stored in the client's browser (Google Chrome), if the client were to open up another tab and try to access Gmail again, then they would not need to log in a second time. However, if the client opens up a new browser, say Internet Explorer, and tries to access Gmail, then they would need to log in again.
- We cannot have session data used across browsers. This is due to security issues.
- A session with each client is assigned a Session ID. The Session data is stored on top of cookies and the server signs them cryptographically. For this encryption, a Flask application needs a defined SECRET_KEY. To create your secret key, you need to set app.secret_key to some string that you choose to be your secret key.
  Here is the general syntax to create this secret key:
  app.secret_key = 'your secret key'
  E.g. **app.secret_key = 'Rick'**
- The Session object is a dictionary object containing key-value pairs of session variables and associated values.
  General syntax for creating a Session object:
  Session[variable name] = value
  E.g. **Session['username'] = 'admin'**
- To remove a session variable, use the pop() method on the session object and mention the variable to be removed.
  General syntax: **session.pop(variable name, none)**
  E.g. **session.pop('username', None)**
- To create a logout feature, do something similar to below:

```
@app.route('/logout')
def logout():
  session.pop(variable name, None)
  return redirect(url_for(function_name'))
```

-

Flash:

- You can use the flash() function to display a message on the screen.
- General syntax: **flash(message, category)**
  The message parameter is the actual message to be flashed.
  The category parameter is optional. It can be either 'error', 'info' or 'warning'.
- On the HTML page(s), you can use the get_flashed_messages() function to get the flash message.
- Typically, the HTML side would look something like this:

```
{% with messages = get_flashed_messages() %}
        {% if messages %}
                {% for message in messages %}
                  ...
                {% endfor %}
          {% endif %}
{% endwith %}
```

- First, we get the flash message from get_flashed_messages(). Then, if there are any messages, we loop through all the messages and do something with them.

Building a login page:

- Using the above information, we can create a login page. Consider the code snippets below:

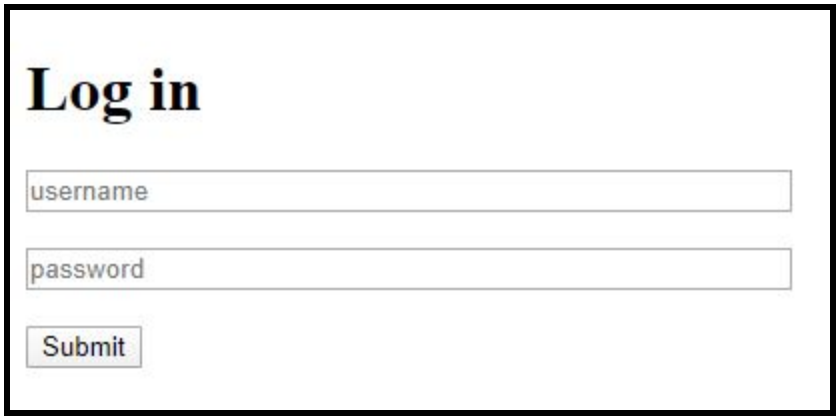
```python
from flask import Flask, request, redirect, render_template, session, url_for, flash

app = Flask(__name__)

# When the user goes onto the home page, they will be redirected to the login page.
@app.route('/')
def index():
    return redirect(url_for('login'))

# The login page.
@app.route('/login', methods=["GET", "POST"])
def login():


    # Removes the user id if there is one inside the session.
    # This way, if the user goes to the login page from any other page, they must login again.
    session.pop('user_id', None)

    if(request.method == "POST"):

        # Gets the username and password the user enters.
        username = request.form['username']
        password = request.form['password']

        # Compares the username and password the user entered to the correct one.
        if (username == "username" and password == "password"):

            session['user_id'] = username
            session['logged_in'] = True

            # Redirects the user to their home page.
            return redirect(url_for('home'))
        else:

            # If the user enters an invalid username password combination, then this error message will show up.
            flash('You have entered an invalid username and/or password. Please Try Again.')
    return render_template('index.html')
```

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

@app.route('/logout')
def logout():
    # Removes the user's session and redirects them to the login page.
    session.pop('user_id', None)
    return redirect(url_for('login'))

if (__name__ == "__main__"):
    # Creates a secret key.
    app.secret_key = b"secretkey"
    app.run(debug = True)
```

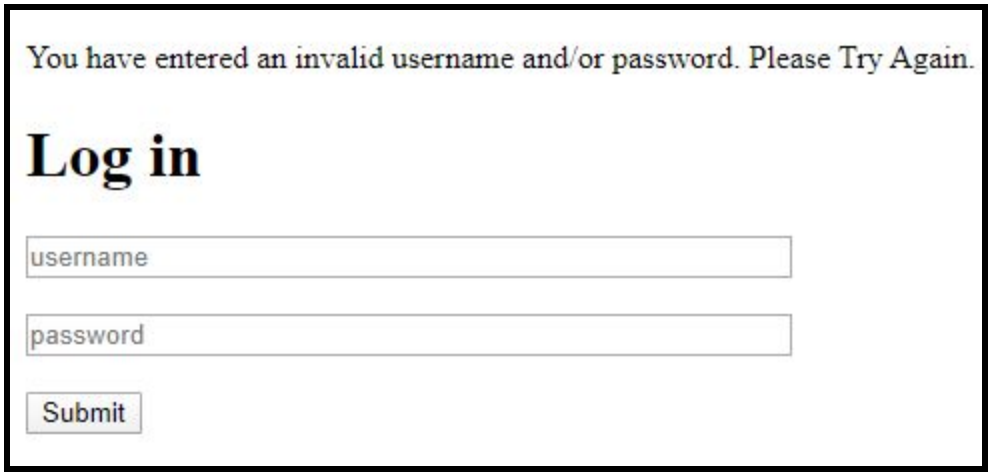
```html
<!DOCTYPE html>
<html>
<head>
    <title> Log In </title>
</head>
<body>
    <!-- This is index.html. -->
    <!-- Prints the error message. -->
    {% with error_message = get_flashed_messages() %}
        {% if error_message %}
            <!-- Loops through the error messages. -->
            {% for error in error_message %}
                <p id="error"> {{ error }} </p>
            {% endfor %}
        {% endif %}
    {% endwith %}

    <!-- Lets the user log in. -->
    <h1> Log in </h1>
    <form action="/login" method="POST">
      <input class="box_content" type="text" name="username" placeholder="username" size=50 required><br> <br>
      <input class="box_content" type="password" name="password" placeholder="password" size=50 required><br> <br>
      <input class="button" type="submit" value="Submit">
    </form>

</body>
</html>
```

```html
<!DOCTYPE html>
<html>
<head>
    <title> Home Page</title>
</head>
<body>
    <!-- This is home.html. -->
    <p> Welcome to your home page! </p>

    <!-- Creates a link that allows the user to log out. -->
    <!-- Note that url_for() takes in the logout function name in login.py. -->
    <a href="{{url_for('logout')}}"> Logout </a>
</body>
</html>
```

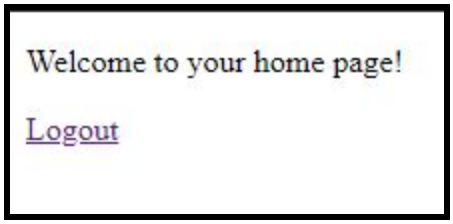
This is what the HTML page looks like:
This is the login screen.



If a user tries to enter an invalid username password combination, then the flash() error message will appear:

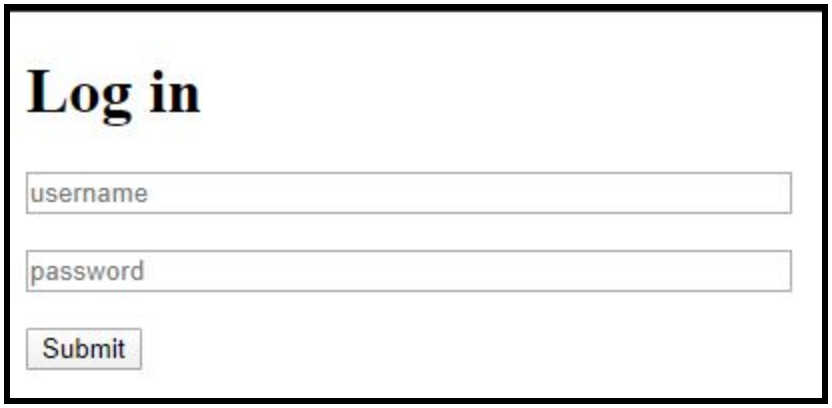


If the user does enter in a valid username password combination, then they will be redirected to the home page:



If the user clicks on the Logout link, they will be logged out and will return to the login

page.



SQLite:

- **Setting up the database:**

```
import sqlite3
from flask import Flask, render_template, request, g

# the database file we are going to communicate with
DATABASE = '/path/to/database.db'

# connects to the database
def get_db():
    # if there is a database, use it
    db = getattr(g, '_database', None)
    if db is None:
        # otherwise, create a database to use
        db = g._database = sqlite3.connect(DATABASE)
    return db

# converts the tuples from get_db() into dictionaries
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

# given a query, executes and returns the result
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
```

- Whenever we want to query information from the database, we need to set up a connection with the database which is done using the function **get_db()**. This function returns a database object of which we can apply queries to.
- The function of **make_dicts()** is to turn the data returned as tuples to dictionaries. By default, Flask returns database data as tuples which are an inconvenient data structure to use. This function utilizes the Factory design pattern to generate dictionaries for all the tuples normally returned by Flask. This function is technically not necessary but will make your life infinitely easier in the long run.
  E.g. I have the database called test.db and it contains this information:

Table: test

	FirstName	LastName	Address
	Filter	Filter	Filter
1	Rick	Lan	123 ABC Road
2	ABC	DEF	234 XYZ Drive

Now, consider these 2 functions:

```
@app.route('/')
def home():
    db = get_db()
    info = query_db('SELECT * FROM test', one=True)
    db.close()
    print("I am not using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

```
@app.route('/Test')
def test():
    db = get_db()
    db.row_factory = make_dicts
    info = query_db('SELECT * FROM test', one=True)
    db.close()
    print("I am using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

This is what you see in the terminal:

```
 * Detected change in '/mnt/c/Users/rick/Desktop/app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
I am not using make_dicts here.
('Rick', 'Lan', '123 ABC Road')
127.0.0.1 - - [06/Apr/2020 02:15:45] "GET / HTTP/1.1" 200 -
I am using make_dicts here.
{'FirstName': 'Rick', 'LastName': 'Lan', 'Address': '123 ABC Road'}
127.0.0.1 - - [06/Apr/2020 02:15:47] "GET /Test HTTP/1.1" 200 -
```

Notice that the line after “I am not using make_dicts here” in the terminal contains the data in a tuple while the line after “I am using make_dicts here” contains the data in a dictionary with the name of the column being the key.

Note: The part of “one=True” means that you are only getting the first row that satisfies the query. In my database above, there are 2 rows, but only the first got printed. If you do not put the “one=True” part, then you will get all the row(s) that satisfies the query. Furthermore, the data will be returned in either an array of tuples, if you do not use make_dicts, or an array of dictionaries, if you do use make_dicts. Lastly, it doesn’t matter how many rows are getting returned, the data will always be inside an array.

If I refresh my two web pages and remove the “one=True” parts, as shown below,

```
@app.route('/')
def home():
    db = get_db()
    info = query_db('SELECT * FROM test')
    db.close()
    print("I am not using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''

@app.route('/Test')
def test():
    db = get_db()
    db.row_factory = make_dicts
    info = query_db('SELECT * FROM test')
    db.close()
    print("I am using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

I get the following output in the terminal:

```
 * Detected change in '/mnt/c/Users/rick/Desktop/app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
I am not using make_dicts here.
[('Rick', 'Lan', '123 ABC Road'), ('ABC', 'DEF', None)]
127.0.0.1 - - [06/Apr/2020 02:27:59] "GET / HTTP/1.1" 200 -
I am using make_dicts here.
[{'FirstName': 'Rick', 'LastName': 'Lan', 'Address': '123 ABC Road'}, {'FirstName': 'ABC', 'LastName': 'DEF', 'Address': None}]
127.0.0.1 - - [06/Apr/2020 02:28:02] "GET /Test HTTP/1.1" 200 -
```

Notice that now, all the returned data is in an array.

Now, if I change the query such that none of the rows will satisfy it, like such,

```
@app.route('/')
def home():
    db = get_db()
    info = query_db('SELECT * FROM test where FirstName=?', ["ooo"])
    db.close()
    print("I am not using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

then the output changes to

```
 * Detected change in '/mnt/c/Users/rick/Desktop/app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
I am not using make_dicts here.
[]
127.0.0.1 - - [06/Apr/2020 02:30:43] "GET / HTTP/1.1" 200 -
```

Notice how that even though none of the rows matched, I still got back an array.

- The function of **query_db()** is to query the database based on a given query. Given a query in the form of a string, it executes and returns the result of the query. In conjunction with make_dicts(), this return will be in the form of a dictionary. query_db() uses a database construct called a cursor to select the data based upon the query and then packages the result to finally return. 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.
- **Database teardown:**
- Whenever you close the Flask application, you need to close any open connections you have to the database. Otherwise, the next time you access the database, you may be blocked by a connection that was still left open. To do this, we create a function that will be called on application teardown that will destroy any currently open connections.
- You can use the function below to close the database.

```
# this function must come after the instantiation of the variable app
# (i.e. this comes after the line app = Flask(__name__))
@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        # close the database if we are connected to it
        db.close()
```

- **Querying Data:**
- Here are the generic steps you can use to query data:
    1. Create a database instance using get_db().
    2. Apply make_dicts to convert the tuples to dictionaries.

3. Get any request parameters (if there are any).
4. Call and execute the database query while storing it somewhere.
5. Close the database.
6. Return the results.

- **Querying without request parameters:**
  Consider the code snippets and database below:

```python
@app.route('/')
def root():
    db = get_db()
    db.row_factory = make_dicts

    # Querying without using request parameters.
    student = query_db('select * from Students')
    db.close()
    # Sending the result of the query to index.html.
    return render_template('index.html', students=student)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title> Showing Querying Databases </title>
</head>
<body>

    <!-- This is how you iterate through an array of dictionaries using Flask. -->
    {% for student in students %}

        <!-- Each "student" is a dictionary. This is how we iterate through dictionaries using Flask. -->
        {% for key, value in student.items() %}

            <!-- This is how you extract the relevant information from the dictionary. -->
            <p> {{key}}: {{value}} </p>
        {% endfor %}
        <br>
    {% endfor %}
</body>
</html>
```

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	1
2	ABC	DEF	2
3	GHI	JKL	3

The HTML output looks like this:

FirstName: Rick

LastName: Lan

StudentNumber: 1

FirstName: ABC

LastName: DEF

StudentNumber: 2

FirstName: GHI

LastName: JKL

StudentNumber: 3

- **Querying with request parameters:**

Consider the code snippers and database below:

```python
@app.route('/<name>')
def getstudent(name):
    db = get_db()
    db.row_factory = make_dicts

    # Querying using request parameters.
    student = query_db('select * from Students where FirstName=?', [name], one=True)
    db.close()
    # Sending the result of the query to index.html.
    return render_template('student.html', students=student)
```

```
<!DOCTYPE html>
<html>
<head>
    <title> Showing Querying Databases </title>
</head>
<body>

    <!-- Each "student" is a dictionary. This is how we iterate through dictionaries using Flask. -->
    {% for key, value in students.items() %}

        <!-- This is how you extract the relevant information from the dictionary. -->
        <p> {{key}} {{value}} </p>
    {% endfor %}

</body>
</html>
```

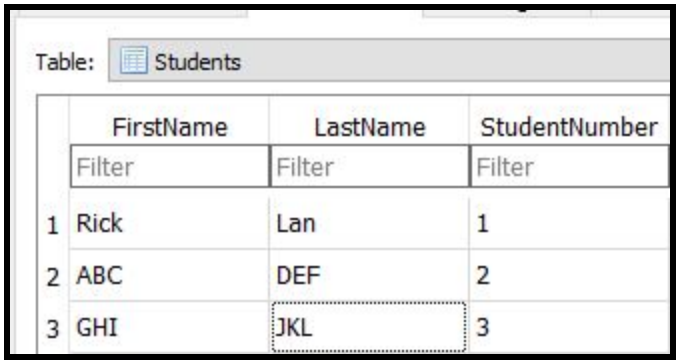
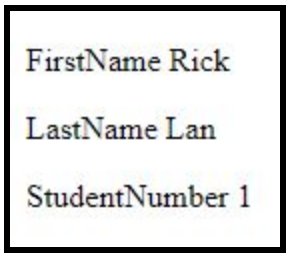


Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	1
2	ABC	DEF	2
3	GHI	JKL	3

This is what the HTML output looks like:



FirstName Rick

LastName Lan

StudentNumber 1

Note: After every query, when we are completely done with the connection, you must close the connection. If you do not close the connection, the next time you access the database, you may get locked out.

Note: We replace the request parameter using a ? instead of something like **query_db('select * from items where name = {name}'.format(name=name))**. This is because using the .format() method to replace parameters leaves the code vulnerable to **SQL injection**. An SQL injection is when a malicious user writes SQL code in the text box which could cause very bad things to happen. Using the ? sanitizes the input and makes it safe to execute.

- **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 you modified data.
    3. Close the cursor.

Note: The reason why you didn't have to create a cursor or use .execute() when you were querying is because the query_db() function did it for you. You just had to call it.

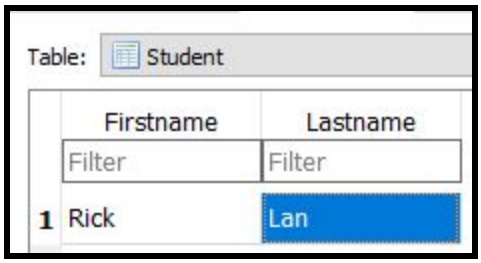
Note: For a SELECT query, the cursor does not modify any data in the database so you will immediately get back the result of the query. For a query that modifies data in the database (such as an INSERT or an UPDATE), the database does not automatically apply the query when it is executed. Instead, it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

- **Inserting into the database:**

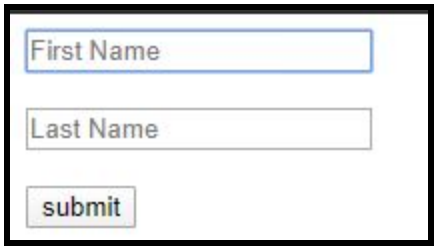
Consider the code snippets and database below:

```python
@app.route('/', methods=["GET", "POST"])
def home():
  if(request.method == "POST"):

    db = get_db()
    db.row_factory = make_dicts
    # make a new cursor from the database connection
    cur = db.cursor()

    # get the post body
    data = request.form

    # Inserts the data into the database
    cur.execute('insert into Student (Firstname, Lastname) values (?, ?)', [
      data['Firstname'],
      data['Lastname']
    ])
    # commit the change to the database
    db.commit()
    # close the cursor
    cur.close()
```

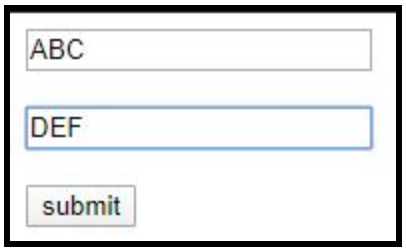
```html
<!DOCTYPE html>
<html>
<head>
    <title> Inserting into Database </title>
</head>
<body>

    <form action="/" method="POST">
        <input type="text" name="Firstname" placeholder="First Name" required><br><br>
        <input type="text" name="Lastname" placeholder="Last Name" required><br><br>
        <input type="submit" value="submit">
    </form>

</body>
</html>
```



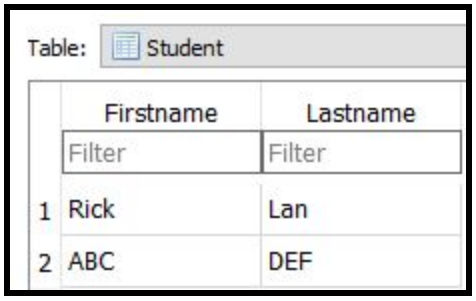
Here is what the HTML page looks like:



If I enter ABC for first name and DEF as last name and click submit:



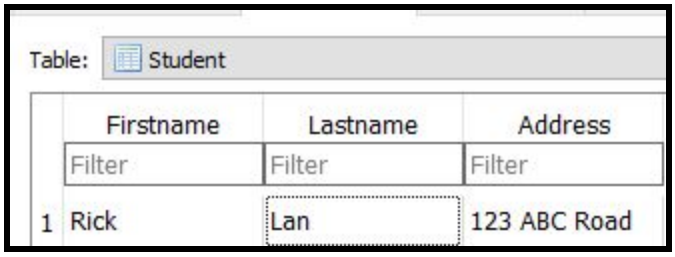
We see the changes in the database:



- **Updating information in the database:**
  Consider the code snippets and database below:

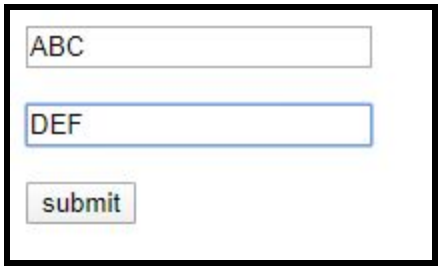
```python
@app.route('/', methods=["GET", "POST"])
def home():
  if(request.method == "POST"):

    db = get_db()
    db.row_factory = make_dicts
    # make a new cursor from the database connection
    cur = db.cursor()

    # get the post body
    data = request.form

    # Updates the data in the db.
    cur.execute('UPDATE Student SET Firstname=?, Lastname=? WHERE Address="123 ABC Road"', [
      data['Firstname'],
      data['Lastname']
    ])
    # commit the change to the database
    db.commit()
    # close the cursor
    cur.close()
```

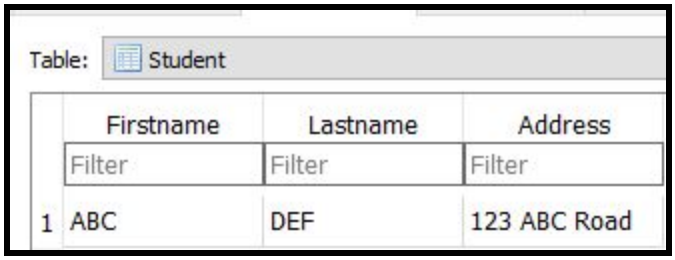
```html
<!DOCTYPE html>
<html>
<head>
    <title> Updating into Database </title>
</head>
<body>

    <form action="/" method="POST">
        <input type="text" name="Firstname" placeholder="First Name" required><br><br>
        <input type="text" name="Lastname" placeholder="Last Name" required><br><br>
        <input type="submit" value="submit">
    </form>

</body>
</html>
```



This is what the HTML page looks like:

If I type ABC for first name and DEF for last name and click submit, the database changes to:





- **Doing a natural join:**
  Consider the code snippets and database below:

```python
@app.route('/')
def home():

    db = get_db()
    db.row_factory = make_dicts

    # Does a natural join.
    natural_join = query_db('SELECT * FROM Student NATURAL JOIN Marks')
    # close the cursor
    db.close()

    return render_template('index.html', info=natural_join)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title> Natural Join </title>
</head>
<body>

    {% for student in info%}
        {% for key, value in student.items() %}
            <p> {{key}}: {{value}} </p>
        {% endfor %}
        <br>
    {% endfor %}

</body>
</html>
```

Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	1	80
2	2	77
3	3	85
4	4	55

Table: Student

	Firstname	Lastname	StudentNumber
	Filter	Filter	Filter
1	ABC	DEF	1
2	GHI	JKL	2
3	MNO	PQR	3
4	STU	VWXYZ	4

If I run the command “select * from Student natural join Marks;” in the database, I get this:

	Firstname	Lastname	StudentNumber	Mark
1	ABC	DEF	1	80
2	GHI	JKL	2	77
3	MNO	PQR	3	85
4	STU	VWXYZ	4	55

When I run app.py, this is what the HTML page looks like:

Firstname: ABC

Lastname: DEF

StudentNumber: 1

Mark: 80

Firstname: GHI

Lastname: JKL

StudentNumber: 2

Mark: 77

Firstname: MNO

Lastname: PQR

StudentNumber: 3

Mark: 85

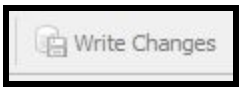
Firstname: STU

Lastname: VWXYZ

StudentNumber: 4

Mark: 55

- **Some common errors:**
  - When you manually update the database, please make sure that you click on the “Write Changes” button. If you do not and your Flask code tries to access the database, you may get errors. When the “Write Changes” button is clicked, it will grey out, as shown here

    This is what the “Write Changes” button looks like when you make change(s) to the database but don’t click it.

    The error looks like this:

  - Do not have spaces in the names of your tables or your columns. When you try to access tables/columns with names that have spaces, it might get mistaken as 2 names rather than 1.

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, shown below:

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. To get the data, passed through the GET request, in Flask, you would need to use **request.args.get()**. Note that by 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, 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. 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 by doing the following:

```
@app.route('/', methods=["GET", "POST"])
def home():
    if (request.method == "POST"):
    ...
```

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) and inside the function, you need to specify that the method you want is POST by doing if(request.method == "POST").
Note: You must put both GET and POST in app.route. If you don't, you may get some error message saying that "Method not allowed."

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.

- Recall that the GET request will show the query in the URL while the POST request hides it. Furthermore, the GET request will cache its data while the POST request won't. For these reasons, and some more, it's best to use the POST request when dealing with sensitive information, such as usernames, passwords, etc.
- **Note:** Do NOT mix the GET request with the POST request. If you do

```
<form action='/' method="POST">
…
</form>
```

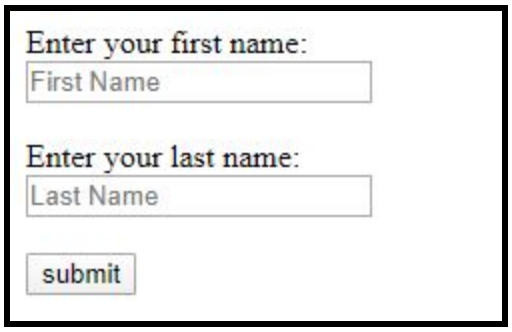
in HTML, you must use request.form in Flask. Using request.get.args will cause an error.

- **Note:** You must include the name attribute in all the places where you want the user to enter information. If you leave out the name attribute, then the data will not be submitted to the backend (Flask).
- **Note:** Say that in your HTML file, you have a <form> element and inside that <form> element, you have an <input> element with the name FirstName. Then, in Flask, you can do **request.form['FirstName']** to get the value associated with FirstName. If you just do **request.form**, it will show you all the values submitted. This is because request.form returns a dictionary with the key being the value of the name attribute. Recall that in Python, to get the value associated with a specific key in a dictionary, you do Dict['key_name'].
- **Note:** If you want to get a specific input from a GET request, use **request.args.get()**, but if you want to get all the data from a GET request, use **request.args**.
- **Note:** The action="..." specifies the endpoint of where this data should go to. When using Flask, it should match the endpoint in app.route(). For example, if you have app.route('/'), then you need to have <form action='/' ...)>.
- Here are some examples of HTTP GET and REQUEST methods:
  E.g. 1. HTTP POST Method
  Consider the pieces of code below:

```
1   from flask import Flask, request, render_template
2
3   app = Flask(__name__)
4
5   @app.route('/', methods=["GET", "POST"])
6   def home():
7       if(request.method == "POST"):
8           print(request.form)
9       return render_template('HTTP_Request_Method.html')
10
11  if (__name__ == "__main__"):
12      app.run(debug = True)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <form action="/" method="POST">

        <label for=FirstName> Enter your first name:</label> <br>
        <input type="text" name="FirstName" placeholder="First Name"> <br>

        <br>

        <label for=LastName> Enter your last name:</label> <br>
        <input type="text" name="LastName" placeholder="Last Name"> <br>

        <br>

        <input type="submit" value="submit">
    </form>
</body>
</html>
```

If the user goes to http://127.0.0.1:5000/, they will see the below picture.



If I enter Rick for First Name and Lan for Last Name and click on the submit button, you will see

```
 * Detected change in '/mnt/c/Users/rick/Desktop/CSCB20 Code/HTTP Request Method/HTTP_Request_Method.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
ImmutableMultiDict([('FirstName', 'Rick'), ('LastName', 'Lan')])
127.0.0.1 - - [06/Apr/2020 17:46:04] "POST / HTTP/1.1" 200 -
```

In the terminal. Furthermore, you will not see the query in the URL.

E.g. 2. HTTP POST Method but with no name attribute.

If I remove the name attribute from First Name, as shown below,

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <form action="/" method="POST">

        <label for=FirstName> Enter your first name:</label> <br>
        <input type="text" placeholder="First Name"> <br>

        <br>

        <label for=LastName> Enter your last name:</label> <br>
        <input type="text" name="LastName" placeholder="Last Name"> <br>

        <br>

        <input type="submit" value="submit">
    </form>
</body>
</html>
```

and I enter Rick for First Name and Lan for Last Name, this is what you see in the terminal:

```
127.0.0.1 - - [06/Apr/2020 17:56:40] "POST / HTTP/1.1" 200 -
 * Detected change in '/mnt/c/Users/rick/Desktop/CSCB20 Code/HTTP Request Method/HTTP_Request_Method.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
ImmutableMultiDict([('LastName', 'Lan')])
127.0.0.1 - - [06/Apr/2020 17:57:16] "POST / HTTP/1.1" 200 -
```

Notice that this time, FirstName isn't shown in the output. This is because if you don't put the name attribute, whatever the user inputs for that textbox/radiobutton/etc will not be submitted to the backend (Flask).

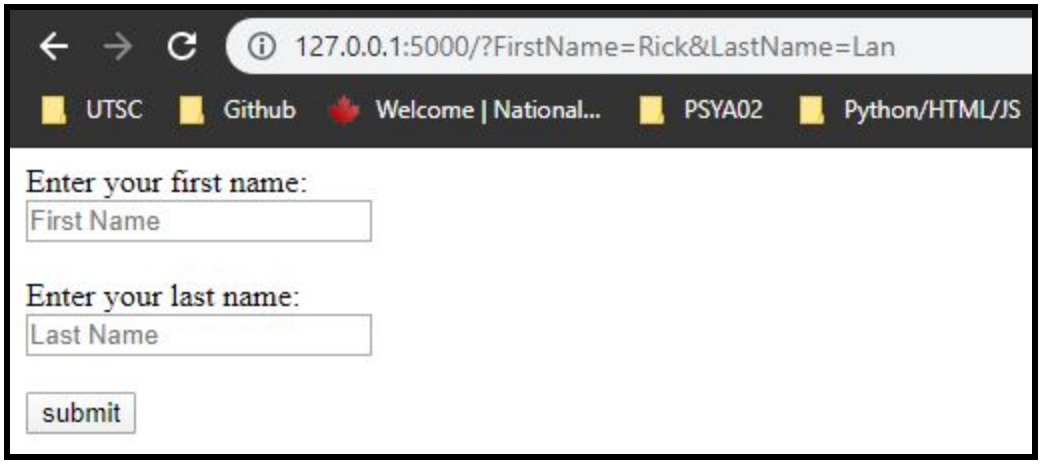
E.g. 3. HTTP GET Request
Consider the following pieces of code below:

```python
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/')
def home():

    # Will print all the data sent via a GET request.
    print(request.args)
    return render_template('HTTP_GET_Method.html')

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

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <form action="/" method="GET">

        <!-- Note: The for attribute of the label must match the name attribute of the input/radiobutton/etc. -->
        <label for=FirstName> Enter your first name:</label> <br>
        <input type="text" name="FirstName" placeholder="First Name"> <br>

        <br>

        <label for=LastName> Enter your last name:</label> <br>
        <input type="text" name="LastName" placeholder="Last Name"> <br>

        <br>

        <input type="submit" value="submit">
    </form>
</body>
</html>
```

The HTML page looks like this:

If I were to type Rick for First Name and Lan for Last Name, I would see this in the URL:



Notice that the query is shown in the URL. Furthermore, this is what I see in the terminal:

```
 * Detected change in '/mnt/c/Users/rick/Desktop/CSCB20 Code/HTTP Get Method/HTTP_GET_METHOD.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
ImmutableMultiDict([('FirstName', 'Rick'), ('LastName', 'Lan')])
127.0.0.1 - - [06/Apr/2020 18:39:09] "GET /?FirstName=Rick&LastName=Lan HTTP/1.1" 200 -
```