Temasek Junior College

2024 JC2 H2 Computing

Web Applications 9 – Handling HTML Form Data

Section	4	Computer Networks
Unit	4.2	Web Applications
Objectives	4.2.3	Use HTML, CSS (for clients) and Python (for the server) to create a web application that is able to: - accept user input (text and image file uploads) - process the input on the local server - store and retrieve data using an SQL database - display the output (as formatted text/ images/table)
	4.2.4	Test a web application on a local server.

1 HTTP Requests

Thus far, we have been getting input from the user by extracting variables from request URLs. However, it is more common to provide input to a web application via a HTML form, where during submission, the browser collects all the input as key-value pairs and encodes it into a single string, which is then sent to the server using a HTTP request.

`GET` requests are used by forms that collect and send parameters for **retrieval of data** from servers. They can be repeated without causing change to server data and state (typically read operations). **`GET`** requests are also used when data needs to be included in the URL for easy bookmarking or sharing. As data appended to the URL is visible in the browser history, server logs and bookmarks, they should be non-sensitive. Data should also meet URL length restrictions of different browsers.[1] Examples of such forms are search forms and filters.

`POST` requests are used by forms that collect and send data to the server, typically during create, update or delete operations, where server state and data will be changed. Instead of the URL, data is included in the body of the **`POST`** request, hidden from view. As such, a **`POST`** request is suitable for handling sensitive data. In general, there is no limit on the size and complexity of data that can be sent in the body of a **`POST`** request, Examples of such forms are registration forms, login forms, and file upload forms.

Some forms are designed such that both the **`GET`** and **`POST`** requests will be made under specific conditions. Some scenarios requiring such a setup include but are not limited to:

- <u>multi-step processes</u> e.g. an initial **`GET`** request to fetch and display data, followed by a **`POST`** request to submit updates or input additional data.

- <u>dynamic data loading</u> e.g. a form might use a **`GET`** request to dynamically load options or data based on user input, followed by a **`POST`** request to submit the final data.

- <u>filtering and submitting data</u> e.g. applying filters to retrieve data fulfilling certain specifications, then submitting final selections from the filtered listing.

- <u>pre-filling forms</u> e.g. retrieving existing data to pre-fill forms for editing and re-submission.

- multi-page forms e.g. navigating through multi-page forms with intermediate **`GET`** requests and a final **`POST`** submission.

[1] Different browsers have a different maximum allowed length for URLs.

1.1 The `request` Object

In Flask, the **`request`** object can be imported from the **`flask`** module to access data collected from HTML forms. The request object provides several methods to access form data, depending on the type of HTTP request used. They include but are not limited to:

- **`request.args`**
- **`request.form`**
- **`request.files`** (see **Web Applications 10**)

The table below provides an overview of how these methods are used to access the different kinds of data submitted via a HTML form.

Attribute	**`request.args`**	**`request.form`**	**`request.files`**
Contents	Dictionary of field names and their associated values from query portion of URL	Dictionary of field names and their associated values	Dictionary of file upload names and their associated **`FileStorage`** objects
Applicable HTTP request	Usually **`GET`**[2]	**`POST`** only	**`POST`** only
Typical Use	Access form data submitted using **`GET`**	Access form data submitted using **`POST`**	Handle files submitted using **`POST`**
Encoding Enforcement	▨	▨	**`enctype`** attribute of **`<form>`** tag must be specified as **`"multipart/form-data"`**

<u>**`request.method`**</u>
If a webpage has been designed to render differently based on the type of user interaction e.g. accessing the form via URL entry in the browser (typically involves a **`GET`** request) or submitting the form after completing it (typically involves a **`POST`** request), the **`request.method`** property can be used to determine the type of request being made.

[2] Also works with **`POST`** requests if the URL has query portion

2 `GET` Requests

The default behaviour for HTML forms is to submit a **`GET`** request. This means that when the **`method`** attribute is not specified in the **`<form>`** tag, a **`GET`** request is made upon form submission. The encoded form data is then visible in the query portion of the request URL.

For instance, the URL **`http://example.com/hello?name=kushat&age=18`** contains a query with two key-value pairs: one for the key **`name`** and another for **`'age'`**.

Parsing key-value pairs encoded in query strings can be tedious and error-prone. Flask simplifies the task by handling this parsing, allowing the data to be accessed as a dictionary from a **`request`** object that can be imported from the flask module.

Exercise 1A
Create the HTML form template **`form_with_get.html`**.

```
1   <!DOCTYPE html>
2   <html>
3     <head><title>Form using GET method</title></head>
4     <body>
5       <form action="{{ url_for('process_with_get') }}">
6         <p>Enter a phrase: <input name="phrase"></p>
7         <p><input type="submit"></p>
8       </form>
9     </body>
10  </html>
```

At this juncture, observe that in Line **5**, **`url_for()`** was used with a function that is called **`process_with_get`** to generate the path that the form data will be submitted to (destination path for the data).

Exercise 1B
Create the HTML template **`analysis_results.html`**.

```
<!DOCTYPE html>
<html>
  <head><title>String Analysis</title></head>
  <body>
    <p>You entered: {{ phrase }}</p>
     <p>The phrase that you have entered contains
        {{ num_vowels }} vowels(s) and
        {{ num_words }} word(s).</p>
  </body>
</html>
```

The template in **Exercise 1B** shall be used to display the results of processing the data collected by the HTML form designed in **Exercise 1A**, which in the case of the code, is to display the number of vowels and number of words in the submitted name.

We shall now write the server code that will render the templates written in **Exercises 1A** and **1B**.

Exercise 1C

Write server code **`server.py`** that will render the HTML templates in **Exercises 1A** and **1B** appropriately.

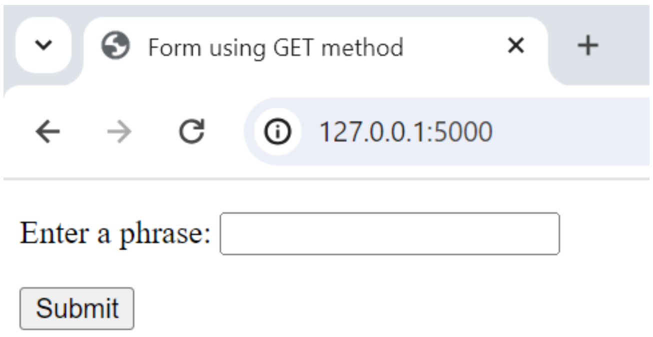
```python
import flask
from flask import render_template, request

app = flask.Flask(__name__)

@app.route('/')
def form():
  return render_template('form_with_get.html')

# Destination path for data that is submitted
@app.route('/process/')
def process_with_get():

  # If phrase is submitted (includes empty submission)
  if 'phrase' in request.args:

    phrase = request.args['phrase'] # Retrieve input as phrase
    in_phrase = phrase.lower() # Lower case for checking vowels

    # Algorithm to count the number of vowels
    vowel_chklst = ['a', 'e', 'i', 'o', 'u']
    num_vowels = 0
    for vowel in vowel_chklst:
      num_vowels += in_phrase.count(vowel) # Number of each vowel

    # Split phrase into the list
    # Length of list equal to number of words in phrase
    num_words = len(in_phrase.split())

    return render_template('analysis_results.html', \
                           phrase = phrase, \
                           num_vowels = num_vowels, \
                           num_words = num_words) \

  # Phrase not submitted, page accessed directly instead
  return 'No form data found!'

if __name__ == '__main__':
  app.run()
```

For the Flask application written in **Exercise 1C:**

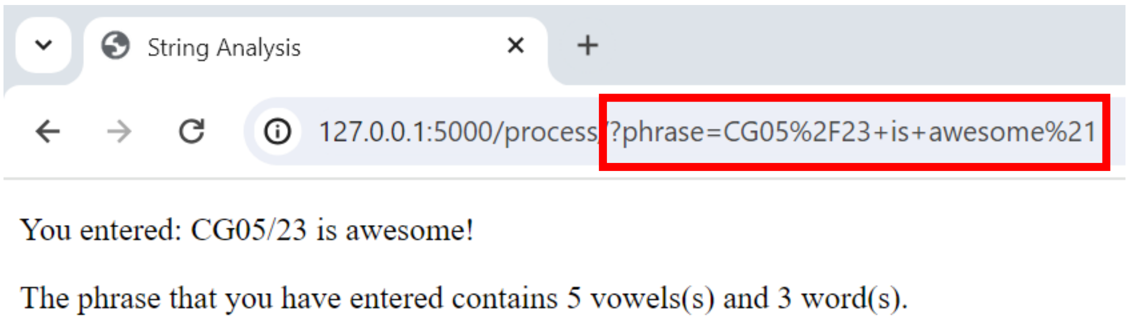
- The **`request`** object is imported from the **`flask`** module in line **`2`**.
- The submitted value of **`phrase`** is accessed via the **`request.args`** dictionary in line **`19.`** This is consistent as the form uses a **`GET`** request by default, since the **`method`** attribute was not specified in the **`<form>`** tag in line **`5`** of Exercise 1A, when was form was created.
- There are two routes:
    - **`/`** renders the HTML form in **Exercise 1A**.
    - **`/process/`** is associated with the **`process_with_get()`** function, which determines whether the path was accessed by:
        - submitting a response through the HTML form created in **Exercise 1A**, or
        - entering the URL directly in the browser

      then renders the results page accordingly.

Visiting **`http://127.0.0.1:5000/`** loads the webpage with an empty form.



Fill up the form with a phrase and click on the **`Submit`** button. Upon submission, observe that the phrase entered is encoded in the resulting URL. This is an implication of using the **`GET`** method.



You entered: CG05/23 is awesome!

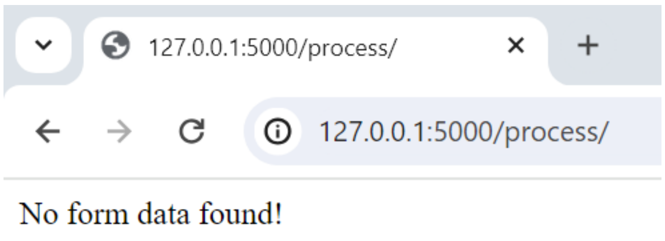
The phrase that you have entered contains 5 vowels(s) and 3 word(s).

The phrase (**`CG05/23 is awesome!`**) being submitted will be stored as with the key **`phrase`** in the **`request.args`** dictionary, as specified by the **`<input name="phrase">`** tag in Line **`6`** of **Exercise 1A**. This allows the phrase to be accessed in the Flask program via the dictionary syntax **`request.args['phrase']`**.

In the Flask program, the access method is not specified in the route to **`/process/`**. As such it can be accessed by default with a **`GET`** request. (See **Web Applications 6**, **Exercise 8** on how to restrict the access to **`POST`** only).

This means that **`http://127.0.0.1:5000/process/`** is directly accessible by entering the URL in the browser. However, accessing the page directly will mean that no data has been submitted via the HTML form.

In doing so, the **`request.args`** dictionary will not have **`phrase`** stored within it. The **`if`**-conditional in Line **`15`** of **Exercise 1C** will be evaluated as **`False`** and the statements within it will not run. Instead, the **`return`** statement in Line **`36`** runs and the message **`'No form data found!'`**



No form data found!

3 `POST` Requests

Submitting form data using a `GET` request may incur several disadvantages:

- Submitted data is appended in the resulting URL, which lets anyone viewing the browser history obtain the data that was sent. This may not be appropriate especially if the data is sensitive.

- **`GET`** requests are not supposed to make changes to the server's data. Using data submitted with a **`GET`** request to perform create, update or delete operations on a database violates HTTP standards.

- Browsers and server software tend to have limits to the length of allowed URLs. Excessively long form data submitted using **`GET`** may become truncated.

These disadvantages can be circumvented by configuring HTML forms to submit the data using **`POST`** requests instead. This can be achieved by setting the **`method`** attribute of the **`<form>`** tag to **`post`** or **`POST`**.

Using a different HTTP request method allows distinction between requests made by clicking a link or entering the URL on a browser, as opposed to requests made by submitting a form.

An advantage of this is that the same URL can be used for both displaying the form, as well as processing the data submitted via the form.

Exercise 2A
Create the HTML template **`form_with_post.html`**.

```
1   <!DOCTYPE html>
2   <html>
3     <head><title>POST Form</title></head>
4     <body>
5       <form method="post">
6         <p>Enter a phrase: <input name="phrase"></p>
7         <p><input type="submit"></p>
8       </form>
9     </body>
10  </html>
```

Contrast Line **5** in **Exercise 2A** with Line **5** in **Exercise 1A**

Notice that in **Exercise 2A**, the **`method`** attribute of the **`<form>`** tag has been specified as **`post`** and the **`action`** attribute of the **`<form>`** tag has been omitted. This will allow the form to be submitted to the same URL used to generate the form.

Exercise 2B
Modify **`server.py`** written in **Exercise 1C** to combine both routes (**`/`** and **`/process/`**) and distinguish between the situations of rendering the empty form and rendering the results page by looking at the HTTP request method used. Save the modified program as **`server_post.py`**.

Recall that the HTTP request method used can be obtained via the **`request.method`** property.

```
1   import flask
2   from flask import render_template, request
3   
4   app = flask.Flask(__name__)
5   
6   @app.route('/', methods = ['GET', 'POST'])
7   def index():
8   
9     # Case where form is accessed directly via URL entry in browser
10    # A GET request is made i.e. request.method == GET
11    if request.method == 'GET':
12      return render_template('form_with_post.html')
13  
14    # Case where form with <form method="post"> is submitted
15    # A POST request is made i.e. request.method == POST
16    if 'phrase' in request.form:
17      phrase = request.form['phrase'] # Retrieve input as phrase
18      in_phrase = phrase.lower() # Lower case for checking vowels
19  
20      # Algorithm to count the number of vowels
21      vowel_chklst = ['a', 'e', 'i', 'o', 'u']
22      num_vowels = 0
23      for vowel in vowel_chklst:
24        num_vowels += in_phrase.count(vowel)
25  
26      # Split phrase into the list
27      # Length of list equal to number of words in phrase
28      num_words = len(in_phrase.split())
29  
30      return render_template('analysis_results.html',
31                             s=s,
32                             num_vowels=num_vowels,
33                             num_words=num_words)
34  
35    # Case of incomplete or invalid data or invalid request
36    return 'No form data found!'
37  
38  if __name__ == '__main__':
39    app.run()
```

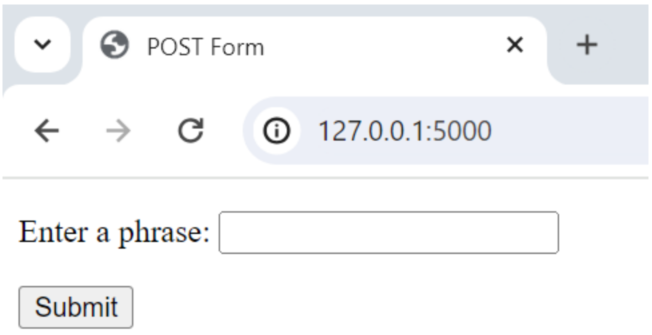
Line **6** of **Exercise 2B** shows how the two routes (**`/`** and **`/process/`**) in **Exercise 1C** has been combined into a single route **`/`**. By specifying the argument to be **`['GET', 'POST']`** for the **`methods`** parameter, the route will work for both **`GET`** and **`POST`** requests. Recall that when the **`methods`** parameter is not specifically defined, the route works for **`GET`** requests by default.

Now that the two routes in **Exercise 1C** has been combined into one, a single **`index()`** function has been defined in line **`10`** of **Exercise 2B** to contain the code previously written in the **`form()`** and **`process_with_get()`** functions in **Exercise 1C**.

The **`index()`** function contains code to check if the request method is **`GET`** in line **`11`**, and if so, renders the empty form. Otherwise, the request method is assumed to be **`POST`** and the submitted form data is processed instead using code from lines **`16`** to **`36`** as appropriate.

Unlike **`GET`** requests where submitted form data is available via the **`request.args`** dictionary, submitted form data is available via the **`request.form`** dictionary instead when it comes to **`POST`** requests. Hence the value of **`phrase`** is accessed via the **`request.form`** dictionary in line **`17`** of **`Exercise 2B`** instead. Contrast this with line **`17`** of **Exercise 1C**, where the value of **`phrase`** is accessed via the **`request.args`** dictionary for the case of involving a **`GET`** request.

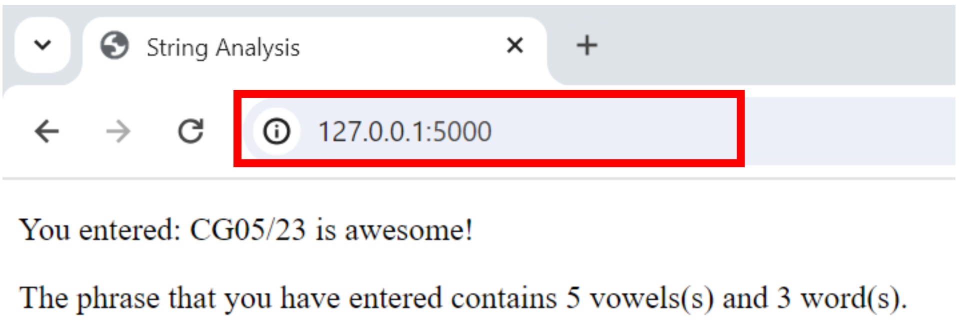
Visiting **`http://127.0.0.1:5000/`** loads the webpage with an empty form.



Enter a phrase:

Submit

Fill up the form with a phrase and click on the **`Submit`** button. The template in **Exercise 1B** will be rendered. Observe that the phrase entered is no longer encoded in the resultant URL. This is an indication that the submission was done using a **`POST`** request. Contrast this with the output obtained previously using a `GET` request (see pg. 5)



You entered: CG05/23 is awesome!

The phrase that you have entered contains 5 vowels(s) and 3 word(s).

While it can be safe to assume that the request method is **`POST`** if the conditional to check whether the method is **`GET`** in line **`11`** evaluates to **`FALSE`**, it is still necessary to check whether the **`request.form`** dictionary contains **`phrase`** stored within it (see line **`17`**)

This is due to the technical possibility of a phony **`POST`** request where data submitted is incomplete or invalid. While the likelihood of the average user sending a phony **`POST`** request is low, it is important for the completeness of the Flask application to be able to handle such a possibility.

In the hypothetical situation where such a phony **`POST`** request occurs, the **`if`** conditional line **`17`** will evaluate to **`FALSE`**, and line **`36`** will run instead.

Challenge

Exercise 3A
Create the following HTML template, **`form.html`**

```
1   <!doctype html>
2   <html>
3     <head>
4       <title>Greeting Form</title>
5     </head>
6     <body>
7       <form method="POST">
8         <input type="text" name="name">
9         <input type="text" name="address">
10        <input type="submit">
11      </form>
12    </body>
13  </html>
```

Exercise 3B
Complete the HTML template **`results.html`** and server code **`app.py`** so that visiting **`http://127.0.0.1:5000/`** will render **`form.html`** and submitting the form will display **`Hello`** ***`name`***, **`your address is:`** ***`address`***, where ***`name`*** is the content of the first text input field and ***`address`*** is the content of the second text input field:

`results.html`

- Complete line **7**

```
1   <!doctype html>
2   <html>
3     <head>
4       <title>Greeting</title>
5     </head>
6     <body>
7       Hello                 ,   your   address   is:
8     </body>
9   </html>
```

`app.py`

- Complete line **2**
- Complete the code from line **6** onwards (You may add in additional lines beyond line **12,** before the code in line **13,** if necessary.**)**

```
1   import flask
2   from flask import
3   app = flask.Flask(__name__)
4   
5   # Use the space below
6   
7   
8   
9   
10  
11  
12  
13  if __name__ == '__main__':
14    app.run()
```