# Recitation – Week 5

PRANUT JAIN

# Plan for today

Flask introduction and setup.

# What is Flask?

Flask is a micro web framework written in Python.

Why a micro web framework ?

-- because it does not require particular tools or libraries.

It does supports extensions like authentication, validation etc.

# Installing Flask

Just run:

pip install Flask

Again if you get errors or exception while installing, try running this command as Administrator(Windows) or Root(Sudo) in Linux.

# Running a simple flask app

**1. Create a script**: (hello.py)

```
from flask import Flask

app = Flask(__name__)


@app.route('/')

def hello_world():

    return 'Hello, World!'
```

2. Run it as

$ export FLASK_APP=hello.py

$ flask run

 * Running on http://127.0.0.1:5000/ or localhost:5000

Use **set** instead of export for Windows.

# Understanding the previous script

imported the Flask class

created an instance of this class.

If you are using a single module (as in this example), you should use __name__ because depending on if it's started as application or imported as module the name will be different ('__main__' versus the actual import name).

use the route() decorator to tell Flask what URL should trigger our function.

# Debugger mode!

The highly useful debugger mode!

The **flask** script is nice to start a local development server, but you would have to restart it manually after each change to your code.

If you enable debug support the server will reload itself on code changes, and it will also provide you with a helpful debugger if things go wrong.

# How to enable debugger mode ?

To enable debug mode you can export the FLASK_DEBUG environment variable before running the server:

$ export FLASK_DEBUG=1

$ flask run

For Windows, use **set** instead of export.

# What does it do ?

it activates the debugger

it activates the automatic reloader

it enables the debug mode on the Flask application.

# Routing

the route() decorator is used to bind a function to a URL. Here are some basic examples:

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

# Variable Rules

To add variable parts to a URL you can mark these special sections as <variable_name>. Such a part is then passed as a keyword argument to your function. Optionally a converter can be used by specifying a rule with <converter:variable_name>.

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username


@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id
```

# Converters

| | |
|---|---|
| *string* | accepts any text without a slash (the default) |
| *int* | accepts integers |
| *float* | like int but for floating point values |
| *path* | like the default but also accepts slashes |
| *any* | matches one of the items provided |
| *uuid* | accepts UUID strings |

# Unique URLs / Redirection Behavior

**Rule1**:

```
@app.route('/projects/')
def projects():
    return 'The project page’
```

**Rule2**:

```
@app.route('/about')
def about():
    return 'The about page’
```

# How are they different ?

Accessing Rule1 without a trailing slash will cause Flask to redirect to the canonical URL with the trailing slash.

Accessing the Rule2 with a trailing slash will produce a 404 “Not Found” error.

This behavior allows relative URLs to continue working even if the trailing slash is omitted, consistent with how Apache and other servers work. Also, the URLs will stay unique, which helps search engines avoid indexing the same page twice.

# Revisiting HTTP methods

GET

The browser tells the server to just get the information stored on that page and send it. This is probably the most common method.

HEAD

The browser tells the server to get the information, but it is only interested in the headers, not the content of the page. An application is supposed to handle that as if a GET request was received but to not deliver the actual content. In Flask you don't have to deal with that at all, the underlying Werkzeug library handles that for you.

POST

The browser tells the server that it wants to post some new information to that URL and that the server must ensure the data is stored and only stored once. This is how HTML forms usually transmit data to the server.

PUT

Similar to POST but the server might trigger the store procedure multiple times by overwriting the old values more than once. Consider that the connection is lost during transmission: in this situation a system between the browser and the server might receive the request safely a second time without breaking things. With POST that would not be possible because it must only be triggered once.

DELETE

Remove the information at the given location.

OPTIONS

Provides a quick way for a client to figure out which methods are supported by this URL. Starting with Flask 0.6, this is implemented for you automatically.

# Handling different HTTP requests

By default, a route only answers to GET requests, but that can be changed by providing the methods argument to the route() decorator.

```
from flask import request


@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```