Introduction to Flask

Cyrus Wong

Source:
https://cs50.github.io/web/
http://www1.cmc.edu/pages/faculty/alee/cs40/lectures/lec15/Lec15x4.pdf

Outline

- Background
- Getting Started
- Basic App Structure
- Template Inheritance
- Jinja2
- Forms
- Cookies
- Sessions
- Debug Mode

Prerequisites

- Knowledge & experience in Python 3.6 or higher!

Web Application (Recap)

- So-called web app
- Is a Software running in web browser
- Created by browser-supported programming language (e.g. JavaScript, HTML and CSS)
- Behaviors depends on how web browsers render

HTTP Protocol (Recap)

HTTP (Hypertext Transfer Protocol) is the system Clients and Servers interact via internet

Normal Flow:

1. URL is entered into a browser of Client
2. Client sends HTTP request to a Server
3. Server interprets the request and sends HTTP response back to Client
4. Client displays the HTTP response in web browser.

What is Flask?

- Microframework for Python
  (keep the core simple but make it extensible)
- Server-side technology
- Based on : Werkzeug + Jinja2

WSGI
(Web Server Gateway interface)

Jinja2
(Template engine)

Features

- Extensible
  (e.g. Flask-Admin, Flask-SQLAlchemy…etc)
- Lightweight to keep the core simple
  - Does what it needs to do and nothing else
- Open source
- Easy to get simple web app running

Why Flask?

- Easy to start
  - Not cover lots of frameworks to start with
- Learn server-side processing concepts
  - Can be applied to real-world applications

Getting started

Environments:

- Installed Python 3.6

https://www.python.org/downloads/

- Created a project directory

```
$ mkdir microblog
$ cd microblog
```

- Created & Activated Virtual Environment

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ _
```

- Installed Flask extension

```
(venv) $ pip install flask
```

Getting started

Environments:

- Installed Python 3.6

https://www.python.org/downloads/

- Created a project directory

```
$ mkdir microblog
$ cd microblog
```

- Created & Activated Virtual Environment

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ _
```

- Installed Flask extension

```
(venv) $ pip install flask
```

Can be all done by CodeSpaces! (IDE)

Getting started

- A sample project “microblog”
- Directory Structure:

```
microblog/
    venv/
    app.py
```

- Flask needs to be told how to import it, by setting the FLASK_APP environment variable:

```
(venv) $ export FLASK_APP=app.py
```

Getting started

- Since environment variables aren't remembered across terminal sessions
- Tedious to set the FLASK_APP environment variable whenever open a new terminal window.
- (Optional) Install the python-dotenv package:

Then Simply write the environment variable name and value to a .flaskenv file in the top-level directory of the project:

.flaskenv: Environment variables for flask command

```
FLASK_APP=app.py
```

Basic App Structure

app.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/")

def index():
    return "Hello, world!"

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

Let’s look into our 1st Flask App

1st Flask App

app.py

```
from flask import Flask
# Import the class `Flask` from the `flask` module, written by someone else.

app = Flask(__name__)

@app.route("/")
def index():
    return "Hello, world!"

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

1st Flask App

app.py

```
from flask import Flask

app = Flask(__name__)
#Instantiate a new web application called `app`, with `__name__
` representing the current file.

@app.route("/")
def index():
    return "Hello, world!"

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

1st Flask App

app.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
# A decorator; when the user goes to the route `/`,
exceute the function immediately below
def index():
    return "Hello, world!"

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

1st Flask App

- Running the Flask App:

```
(venv) $ flask run
    * Serving Flask app "microblog”
    * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

- Once the server starts up, it goes into a loop that waits for requests and services them as they come
- You can stop the loop by hitting Ctrl-C on the running server app

Initialization

- All Flask app must create an application instance like this:

```
from flask import Flask
app = Flask(__name__)
```

- The web server passes all requests coming from clients to this object, using a protocol called Web Server Gateway Interface (WSGI)

View function & Routes

- Clients (web browsers) send requests to the web server
- The server relays the requests to the Flask app instance/object
- The app object needs to know what handler code needs to run for each URL request (e.g., http://xxx…)

View function & Routes

- **<u>View function</u>**: Handler function mapped to URLs (written in Python functions)
  * function Name need to be logical
- **<u>Route</u>**: Association between a URL and its view function
- Sample of '/' is handled by index():

Route: mapping of '/' to index()

URL: '/'

```
@app.route("/")
def index():
    return "Hello, world!"
```

View function: index()

Server Setup

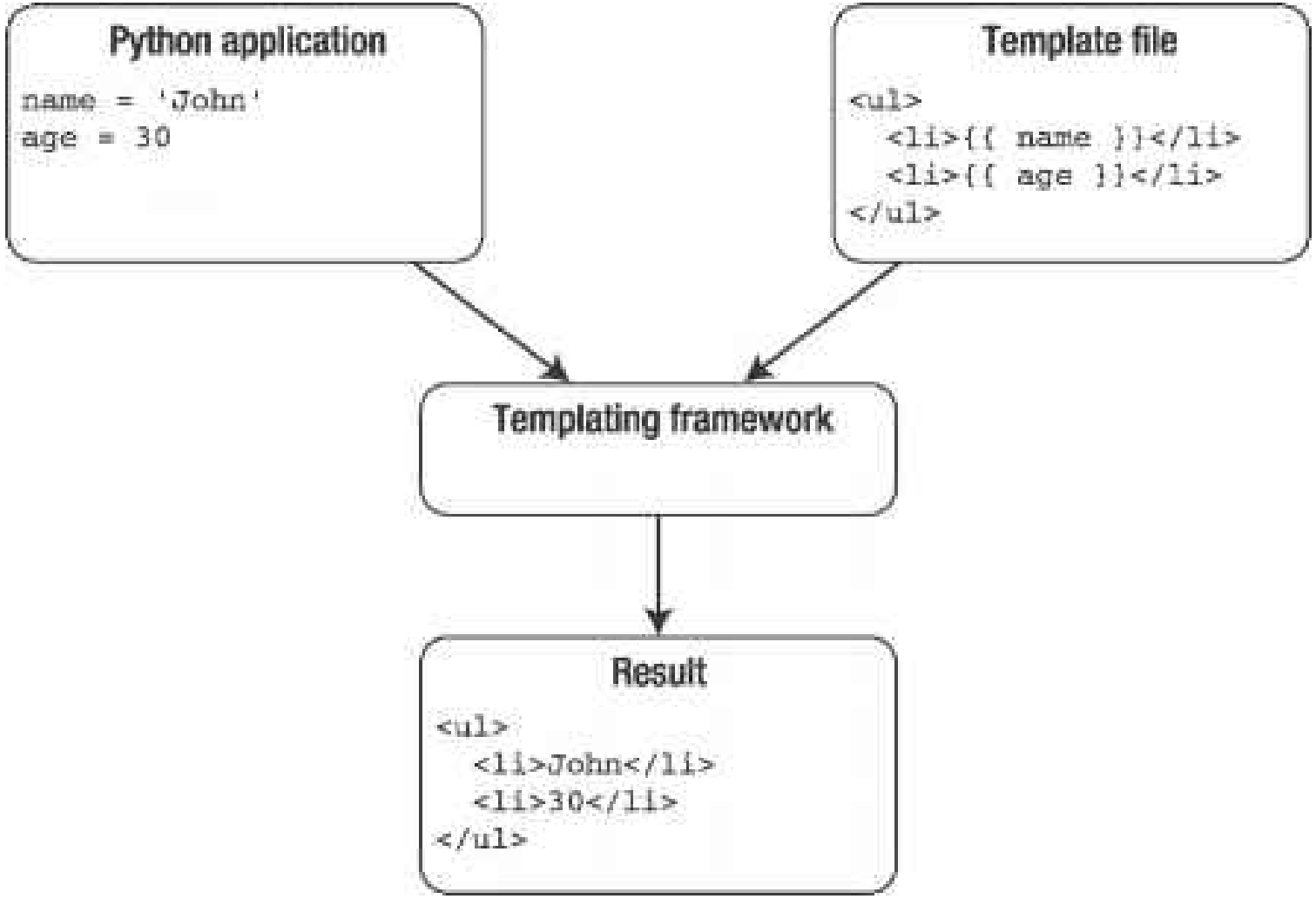
- Server startup The application instance has a run method that launches Flask's integrated development web server:

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

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

This conditional check is a Python idiom that ensures that the development web server is started only when the script is executed directly, i.e., it won't be executed if the script is imported by another script.

Templating System



Template Inheritance

- A web app with numbers of pages, always have common part that will have **duplicate codes**.
- In order to cut down on repetitive HTML amongst many different pages, Jinja2 has a feature called ‘template inheritance’ that uses the idea of blocks to organize content.
- Let’s have look on this example project:

```
inheritance/
  app.py
  templates/
    index.html
    layout.html
    more.html
  venv/
```

Template Inheritance

```
inheritance/
  app.py
  templates/
    index.html
    layout.html
    more.html
  venv/
```

- No content in View Function
- Different routes going to different url :
  http://127.0.0.1:5000/ -> http://127.0.0.1:5000/index.html
  http://127.0.0.1:5000/more -> http://127.0.0.1:5000/more.html

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/more")
def more():
    return render_template("more.html")
```

Invokes Jinja2 templating engine

Template Inheritance

```
inheritance/
  app.py
  templates/
    index.html
    layout.html
    more.html
  venv/
```

- We separate the presentation from the logic by creating a template layout.html inside of templates/.

```
<html>
    <head>
        <title>{% block title %}{% endblock %}</title>
    </head>
    <body>
        {% block body %} {% body %}
    </body>
</html>
```

- Now we have a template called layout.html which has some basic elements but also includes a control statement which allows us to derive and extend the template.

Template Inheritance

```
inheritance/
  app.py
  templates/
    index.html
    layout.html
    more.html
  venv/
```

- Defining first set of content

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

{% block title %}
   First Page
{% endblock %}

{% block body %}
   <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce
placerat rutrum nisi at. </p>
   <a href="{{ url_for('more') }}">See more...</a>
{% endblock %}
```

Template Inheritance

```
inheritance/
  app.py
  templates/
    index.html
    layout.html
    more.html
  venv/
```

- Defining second set of content

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

{% block title %}
   Second Page
{% endblock %}

{% block body %}
   <p>
      Donec lobortis dapibus magna, quis facilisis velit malesuada a.
Nunc iaculis augue nulla, sed sodales.
   </p>
   <a href="{{ url_for('index') }}">Go back</a>
{% endblock %}
```

Jinja2

- When any string is entered as a route, that will be stored as name, which is can then be used inside the decorated function.

```
@app.route("/<string:name>")

def hello(name):
    return f"Hello, {name}!"
```

- Since Python code is rendering the website, anything Python is capable of can be used. For example, name can be capitalized before it’s displayed:

name = name.capitalize()

Jinja2

- HTML can also be used inside the return value:

  return f"<h1>Hello, {name}!</h1>".
- Inline HTML isn’t that useful, though. Separate HTML files can be used like so:

```
from flask import Flask

    app = Flask(__name__)
    @app.route("/")
    def index():
        return render_template("index.html")
```

Jinja2

- Jinja2 templates allow us to do a lot more than just write HTML – we can write extensible HTML.
- with Jinja syntax, the file is still named with .html extension

Jinja2 – arguements

- {{ arg }} corresponds to template **arguments**. We can pass arg=val to our template.
- Multiple routes on the Flask server, can be defined:

  See more...

  more is the name of a function associated with a route.

```
#app.py
headline = "Hello, world!"
return render_template("index.html", headline=headline)

#index.html
<h1>{{ headline }}</h1>
```

Jinja2 - control

- {% %} encompasses control **statements** (if, for, block, etc).

```
#conditional statement
{% if new_year %}
        <h1>Yes! Happy New Year!</h1>
{% else %}
        <h1>No.</h1>
{% endif %}

#for-Loop
{% for name in names %}
        <li>{{ name }}</li>
{$ endfor %}
```

Jinja2 - block

- {% block content %} {% endblock %} identifies a portion of the **<u>html</u>** (called “content”) in which more content could be inserted.

```
{% block body %}
   <p>
      html content is written here.
   </p>
{% endblock %}
```

Jinja2

- index.html and any other template files should be stored in a directory named **templates**.
- Variables can be defined as Python variables in app.py and used in HTML templates by passing them in as arguments to render_template.

Forms

- With Flask and Jinja2, the results from HTML forms can now be actually stored and used.
- An HTML form might look like this:

```
<form action="" method="post">
   <input type="text" name="name" placeholder="Enter Your N
ame">
   <button>Submit</button>
<form>
```

Forms

- The **action** attribute lists the route that should be ‘notified’ when the form is submitted.
- The **method** attribute is how the HTTP request to submit the form should be made. The default method is GET. however, POST should be used.
- The **name** attribute of the input, can be referenced when the form is submitted.

Forms

- The Python code to process the form might look like this:

```
from flask import Flask, render_template, request

@app.route("/hello", methods=["POST"])
def hello():
    name = request.form.get("name")
# take the request the user made, access the form
    return render_template("hello.html", name=name)
# and store the field called `name` in a Python variable also called
`name`
```

Forms

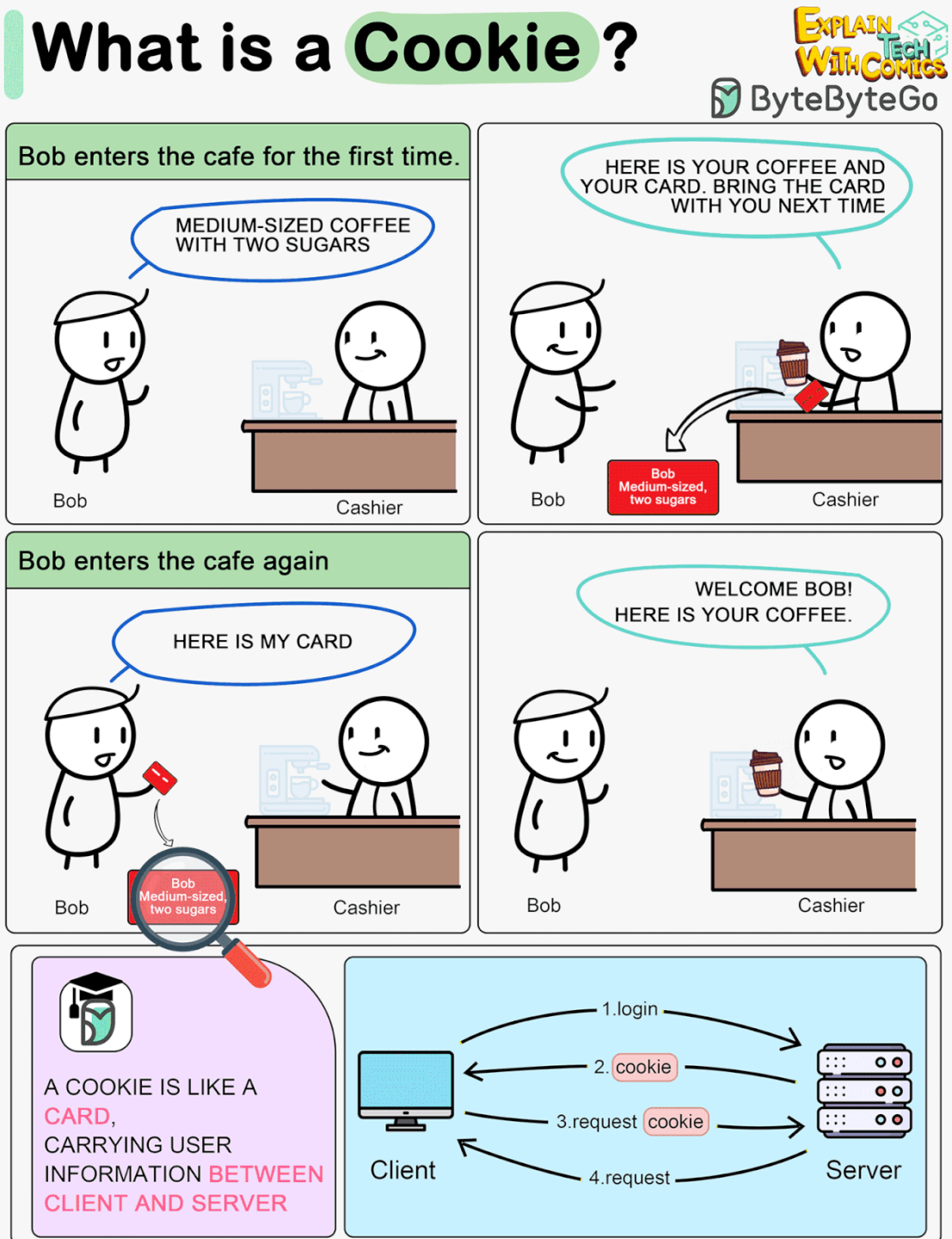
- The route /hello is the same hello listed in the Jinja2 code.

  This route can also accept the POST method, which is how the form's data is being submitted.

- If any other method is used to access this route, a Method Not Allowed error will be raised.
  - Request method type can be checked with request.method, which will be equal to, for example, "GET" or "POST".

Cookies

A cookie is stored on a client's computer in the form of a text file. Its purpose is to remember and track data pertaining to a client's usage for better visitor experience and site statistics.

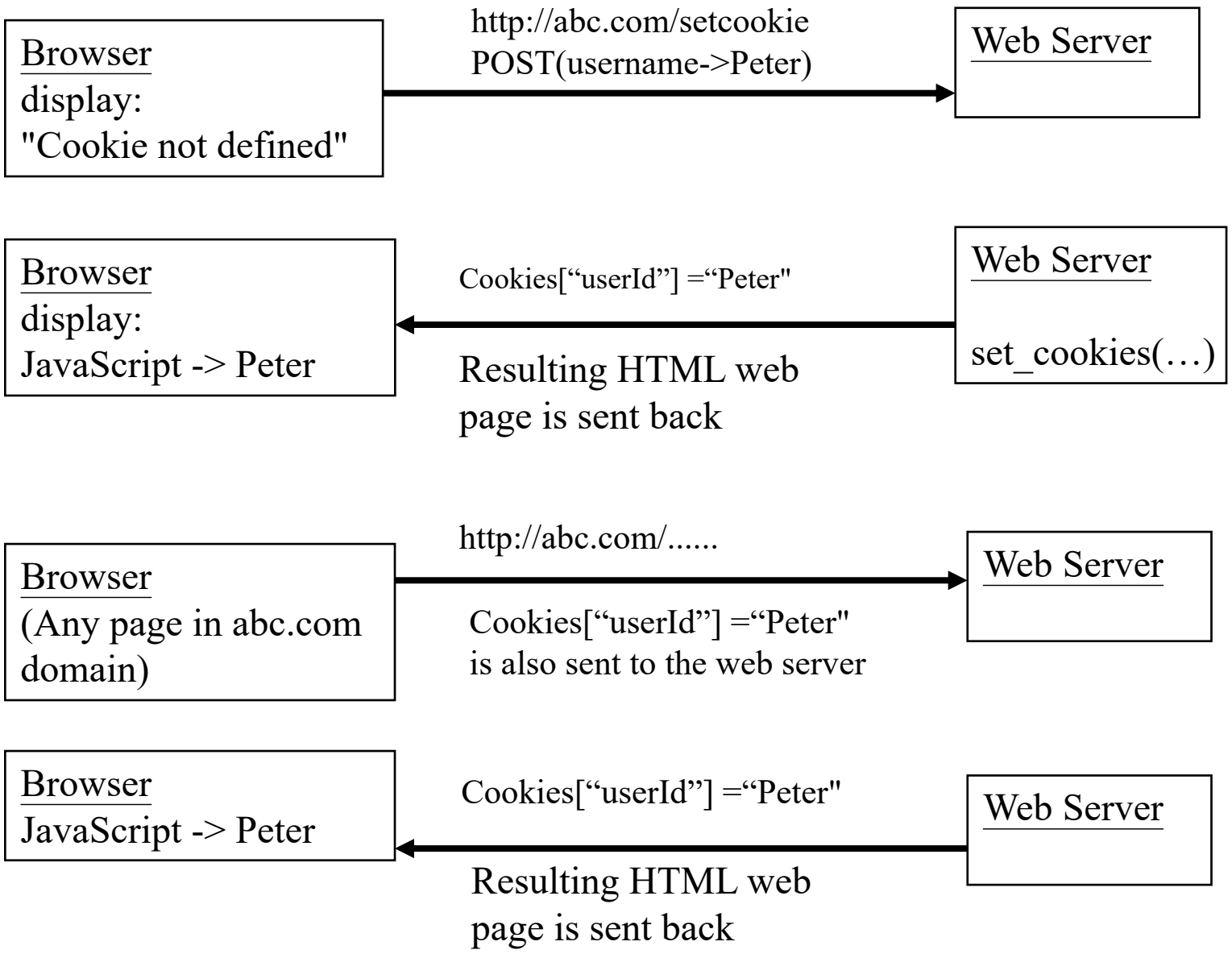


Cookies

set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure=False, httponly=False, samesite=None)

- **key** – the key (name) of the cookie to be set.
- **value** – the value of the cookie.
- **max_age** – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client's browser session.
- **expires** – should be a *datetime* object or UNIX timestamp.
- **path** – limits the cookie to a given path, per default it will span the whole domain.
- **domain** – if you want to set a cross-domain cookie. For example,domain=".example.com" will set a cookie that is readable by the domain www.example.com, foo.example.com etc. Otherwise, a cookie will only be readable by the domain that set it.
- **secure** – If *True*, the cookie will only be available via HTTPS
- **httponly** – disallow JavaScript to access the cookie. This is an extension to the cookie standard and probably not supported by all browsers.
- **samesite** – Limits the scope of the cookie such that it will only be attached to requests if those requests are "same-site".

How Cookies works



Cookies

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

app = Flask(__name__)

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

```html
<html>
    <body>
        <form action = "/setcookie" method = "POST">
            <p><h3>Enter userID</h3></p>
            <p><input type = 'text' name = 'nm'/></p>
            <p><input type = 'submit' value = 'Login'/></p>
        </form>
    </body>
</html>
```

Enter userID

Login

Cookies

```python
@app.route('/setcookie', methods = ['POST', 'GET'])
def setcookie():
    if request.method == 'POST':
        user = request.form['nm']
    resp = make_response(render_template('readcookie.html'))
    expire = datetime.datetime.now()
    expire = expire + datetime.timedelta(seconds=5)
    resp.set_cookie('userID', user, expires=expire)
    resp.set_cookie('secureUserID', user, expires=expire, secure=True)
    resp.set_cookie('httpOnlyUserID', user, expires=expire, httponly=True)
    return resp

@app.route('/getcookie')
def getcookie():
    userID = request.cookies.get('userID') or ''
    secureUserID = request.cookies.get('secureUserID') or ''
    httpOnlyUserID = request.cookies.get('httpOnlyUserID') or ''
    return f"""
<h1>userID: {userID} </h1>
<h1>secureUserID: {secureUserID} </h1>
<h1>httpOnlyUserID: {httpOnlyUserID} </h1>
"""
```

Readcookie.html

```html
<p>Read cookies from JavaScript</p>
<script>
    document.write(document.cookie);
</script>
```

Cookies - Secure & HttpOnly

HTTPS

https://2f183fea

Read cookies from JavaScript

userID=aaaa; secureUserID=aaaa;
AWSALB=aBQK/Cz4UagwwP8C6

setcookie

https://adc576133a5f467e8e

getcookie

userID: aaaa

secureUserID: aaaa

httpOnlyUserID: aaaa

https://adc576133

After 5 seconds

userID:

secureUserID:

httpOnlyUserID:

HTTP

Not secure

Read cookies from JavaScript

userID=aaaa

Not secure | 52.87.219.97:80

userID: aaaa

secureUserID:

httpOnlyUserID: aaaa

Not secure | 52.87.21

userID:

secureUserID:

httpOnlyUserID:

Cookies - Remark

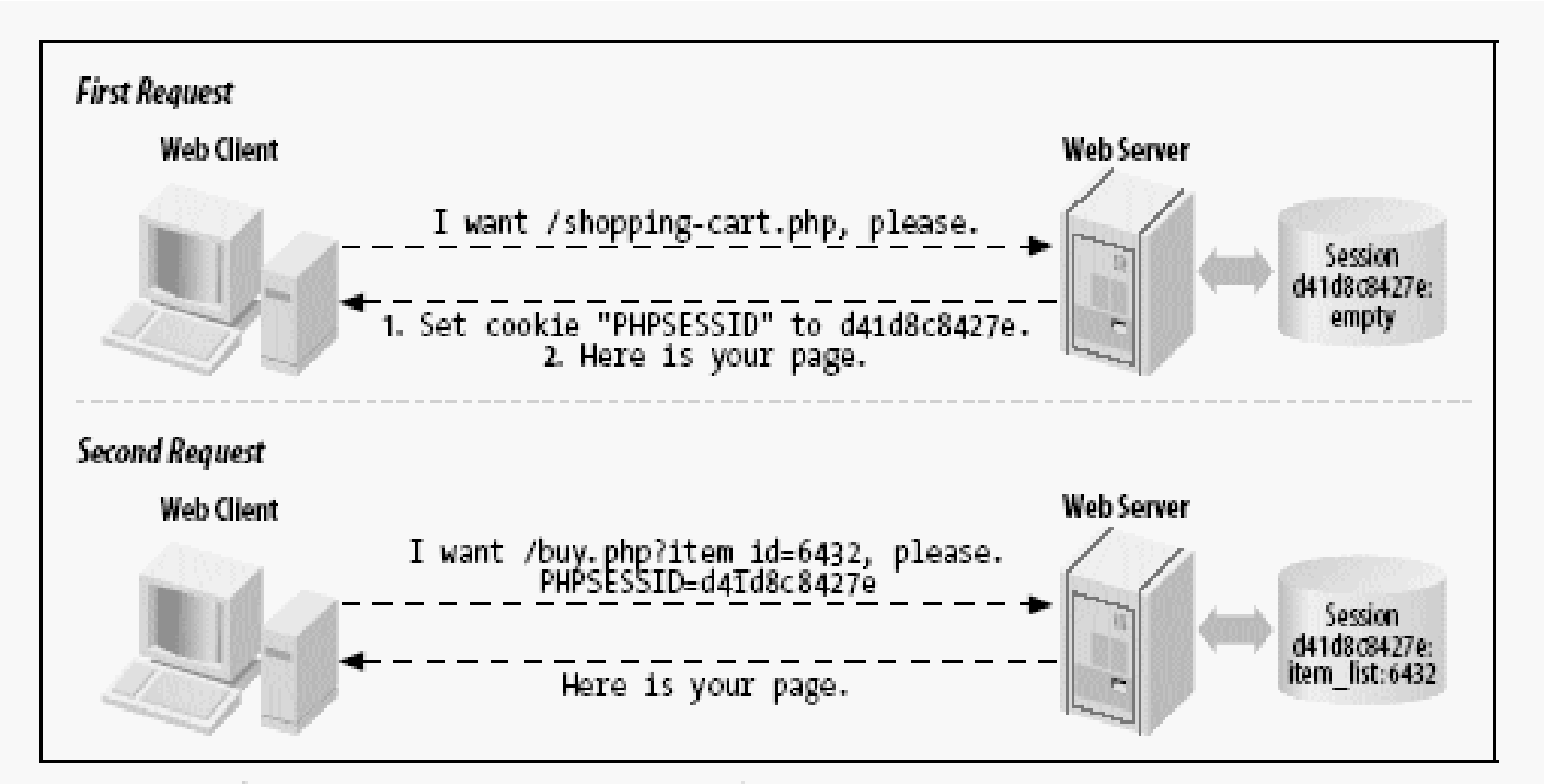
- Delete Cookies
  set_cookie('foo', 'bar', max_age=0)
- Not secure.
- Disabled.
- Each Cookie can store no more than 4KB of data.
- Sent everytime you request a page from the server. 20 cookies and each of them store 4KB of data => additional payload of 80KB on every request!

```
<script>
    document.cookie = "foo=bar;";
    if (!document.cookie)
    {
        alert("This website requires cookies to function properly");
    }
</script>
```

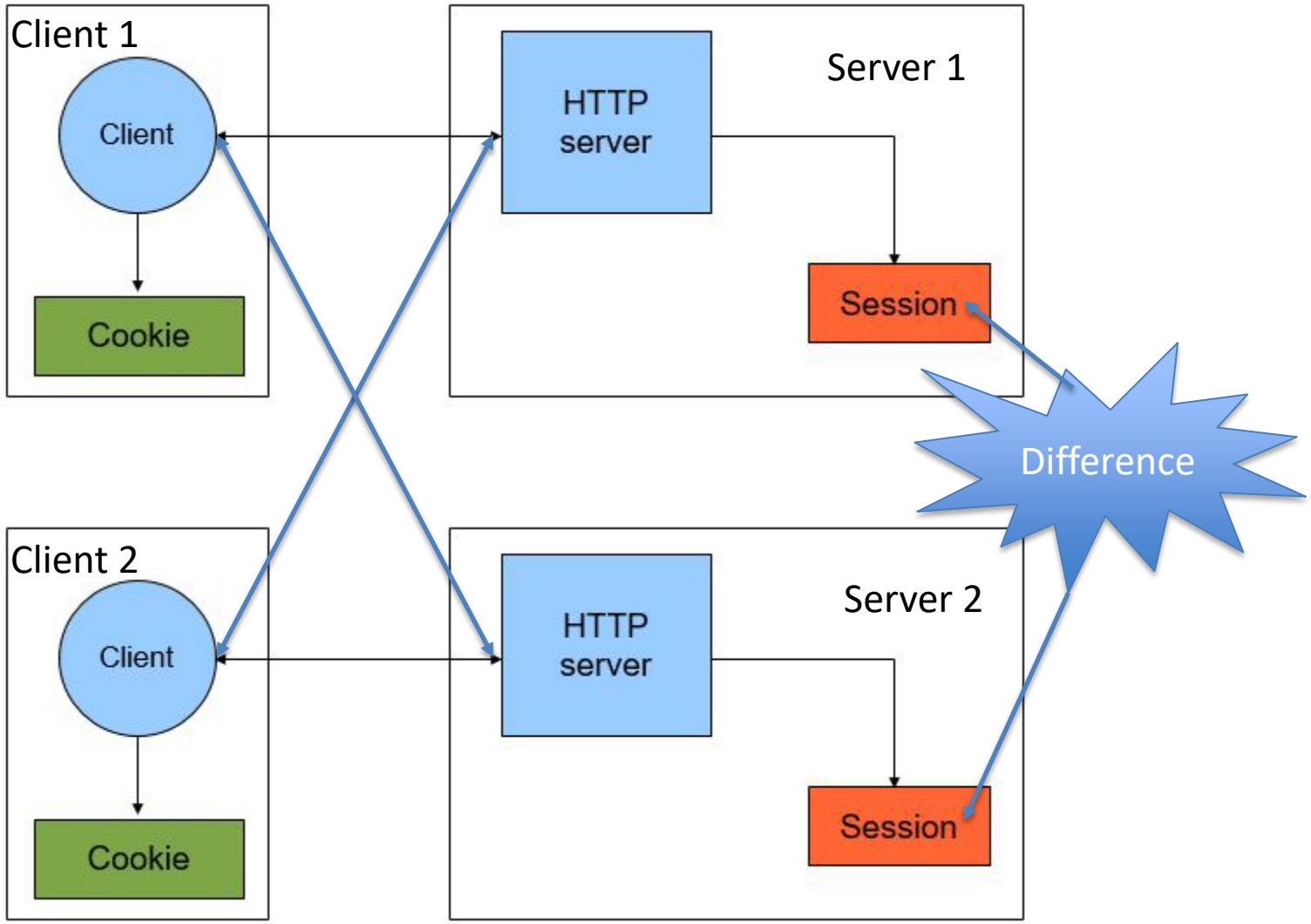
Session

- To use session must set the secret key.
- Session object works like a dictionary.
- Cookie used to store session data!
  (client-side sessions [Default])
  In PHP, session cookie doesn't store session data (server-side session)!
- Can view, but Can't modify! [signed]
  (The difference from Basic Cookie!)
- Use server-side sessions =>
  Flask-Session and Flask-KVSession

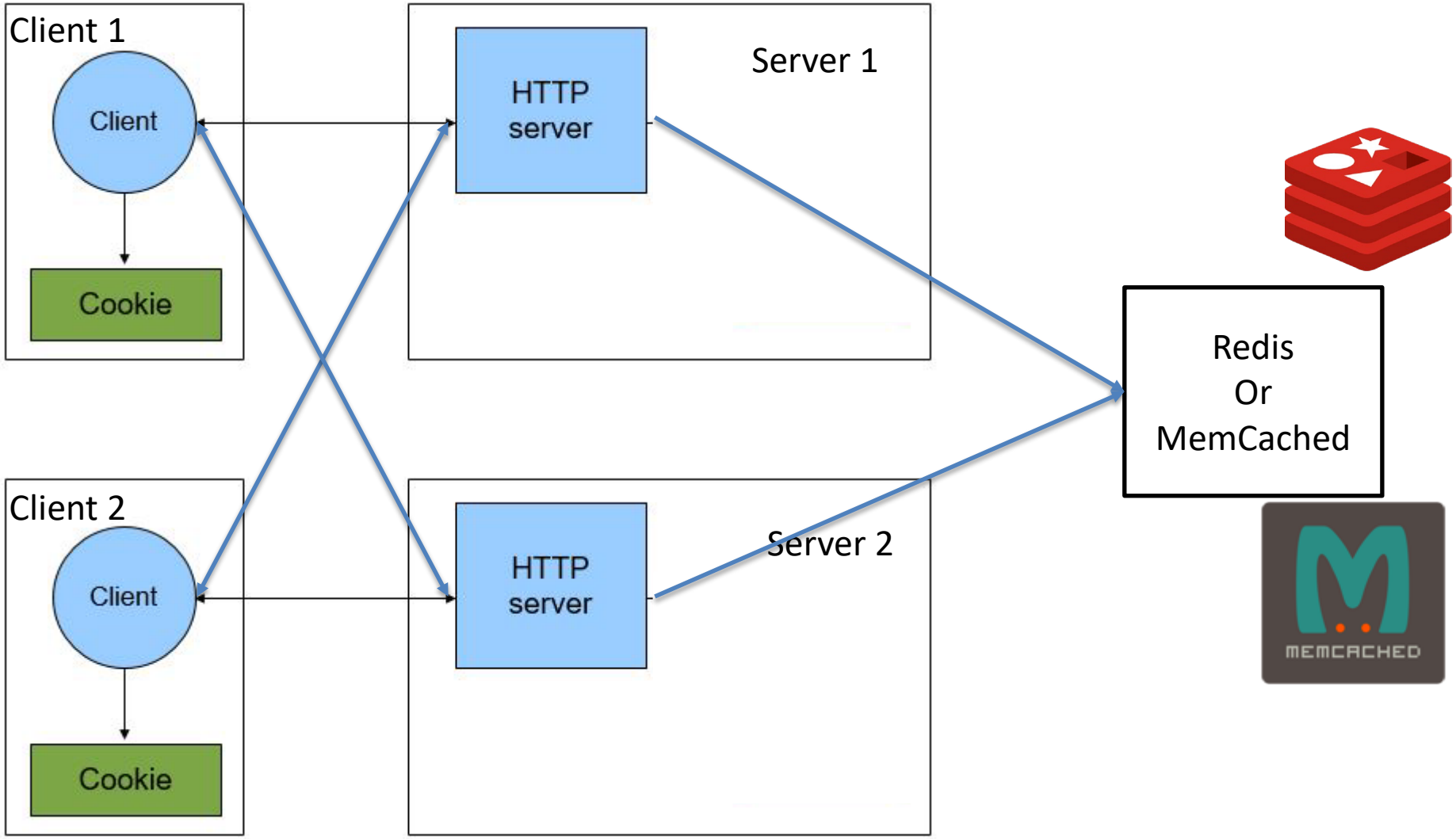
Server-Side Session i.e. PHP



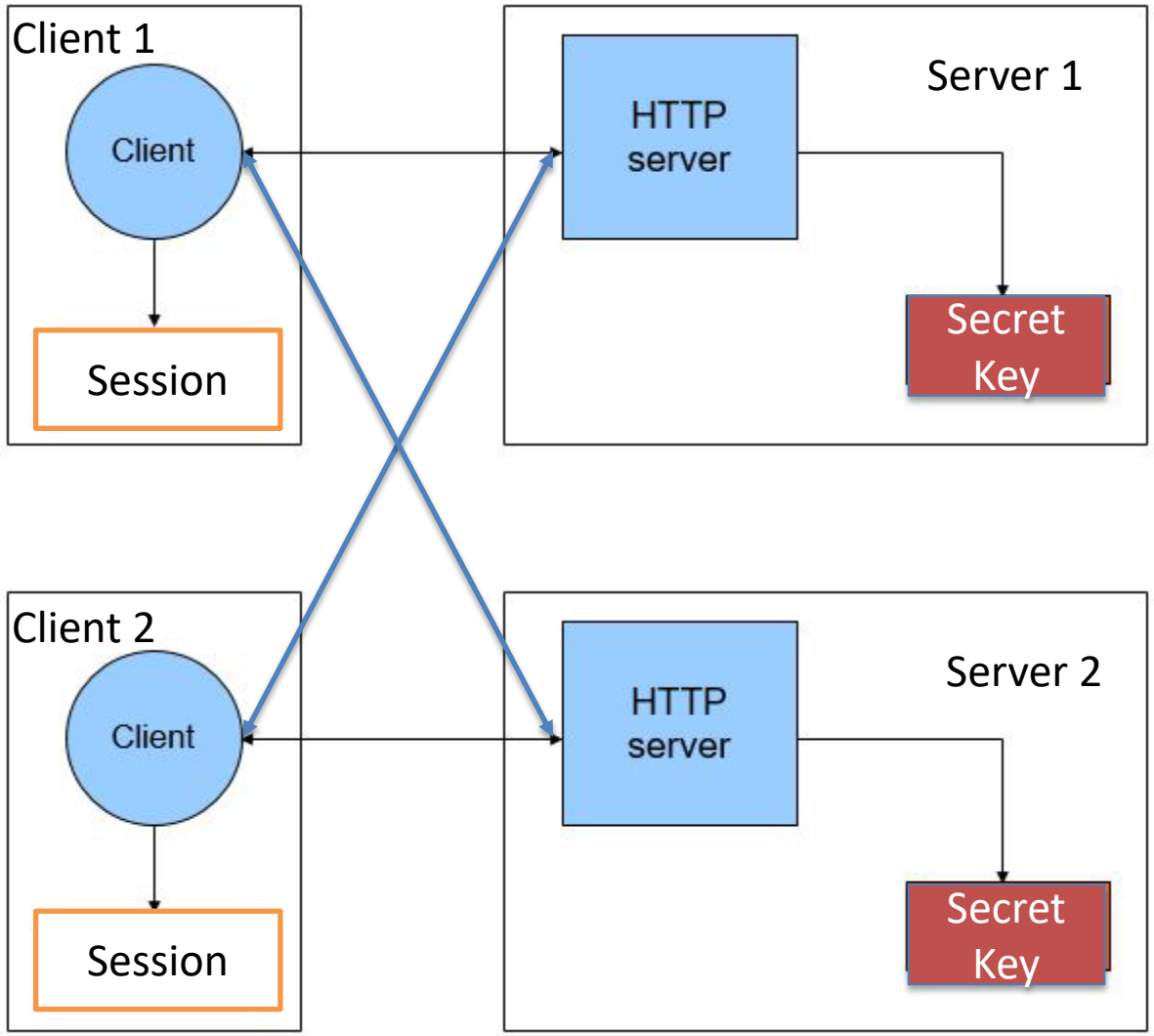
Server-Side Session i.e. PHP



Server-Side Session i.e. PHP



Client-Side Session i.e. Flask



Sessions

- Sessions are how Flask can keep track of data that pertains to a particular user. Let’s take a note-taking app, for example. Users should only be able to see their own notes.
- To use sessions, they must be imported and set up:

```
from flask import Flask, render_template, request, session
from flask_session import Session

config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

Sessions

```
from flask import Flask, render_template, request, session
# gives access to a variable called `session`, which can be used to keep vaules
that are specific to a particular user
from flask_session import Session

config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

Sessions

```
from flask import Flask, render_template, request, session
from flask_session import Session
# an additional extension to sessions which allows them

config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

Sessions

```
from flask import Flask, render_template, request, session
from flask_session import Session

config["SESSION_PERMANENT"] = False
# to be stored server-side  app.config
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

Sessions

Then, assuming there is some HTML form that can submit a note, the note can be stored in a place specific to the user using their session:

```
@app.route("/", methods=["GET", "POST"])
def index():
    if session.get("notes") is None:
        session["notes"] = []
    if request.method == "POST":
        note = request.form.get("note")
    session["notes"].append(note)
    return render_template("index.html", notes=session["notes"])
```

Sessions

- notes is the list where the notes will be stored. If the user doesn’t have a notes list already (checked with if session.get("notes") is None), then they are given an empty one.
- If a request is submitted via "POST" (that is, through the form), then the note is processed from the form in the same way as before.

Sessions

- The processed note, now in a Python variable called note, is appended to the notes list. This list is itself inside a dict called session. Every user has a unique session dict, and therefore a unique notes list.
- Finally, the notelist is rendered by passing session["notes"] to render_template.

Sessions - Remove

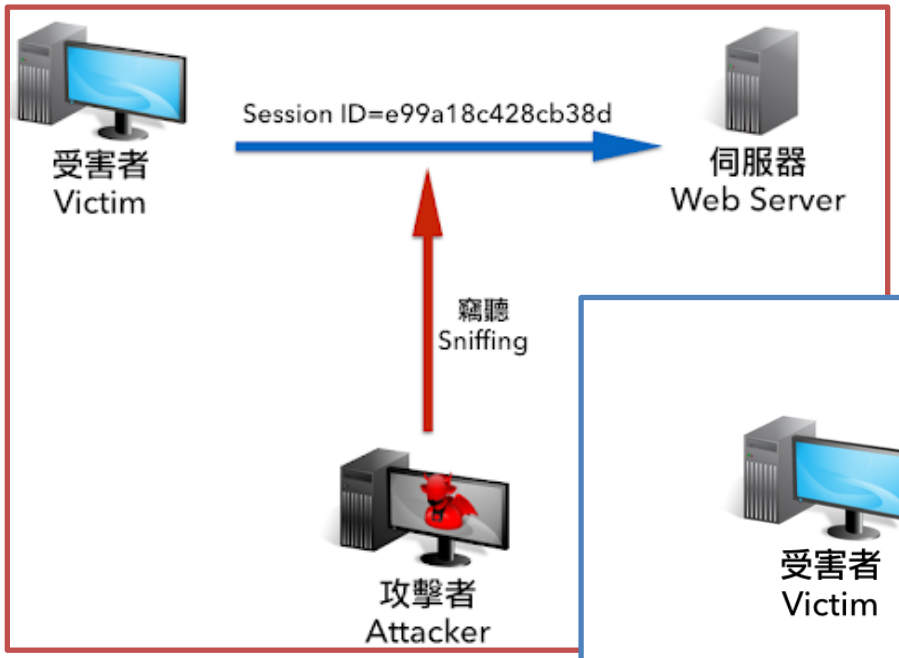
To release a session variable use pop() method.

```
session.pop('username', None)
```

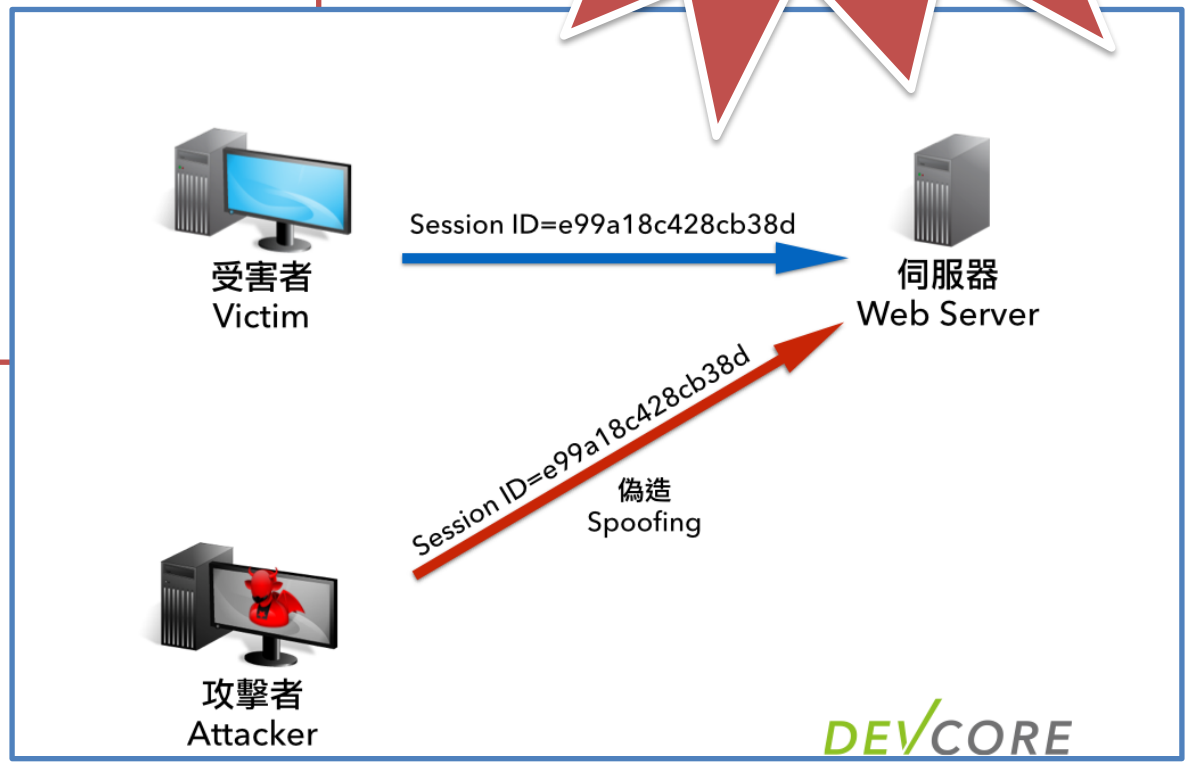
Clearing the contents of a session

```
session.clear()
```

Session Hijacking (竊取 Session ID)



Session Cookie have to use HTTP Only + Secure (Https)



https://devco.re/blog/2014/06/03/http-session-protection/

Flask’s Debug Mode

- After Update source code, Will Flask auto reflect? It depends!
- If Debug mode, YES!
- Else, NO! Need to restart the Flask server
  if you make any modifications to the codes

Flask's Debug Mode

- For development, we can turn on the debug mode, which enables the auto-reloader as well as the debugger.
- There are two ways to turn on debug mode:

1. Set the debug attribute of the Flask instance app to True:

```
app = Flask(__name__)
app.debug = True
# Enable reloader and debugger
......

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

2. Pass a keyword argument debug=True into the app.run():

```
app = Flask(__name__)
......
if __name__ == '__main__':
  app.run(debug=True)  # Enable
```

Flask's Debug Mode

- In debug mode, the Flask app monitors your source code, and reload the source code if any modification is detected (i.e., auto-reloader). It also launches the debugger if an error is detected.
- IMPORTANT: Debug mode should **NOT** be used for production, because it impedes the performance, and worse still, lets users execute codes on the server.

FLASK_DEBUG Environment Variable (Since Flask 0.11)

Starting from Flask 0.11, you can enable the debug mode via environment variable FLASK_DEBUG without changing the codes, as follows:

$ **export FLASK_APP=hello_flask.py**

$ **export FLASK_DEBUG=1**

$ **flask run**