Department of Information Engineering, CUHK
MScIE – 2nd Semester, 2015/16

IEMS 5722
Mobile Network Programming and Distributed Server Architecture

Lecture 5
Web and Application Servers

Lecturer: Albert C. M. Au Yeung

11th February, 2016

HTTP Servers

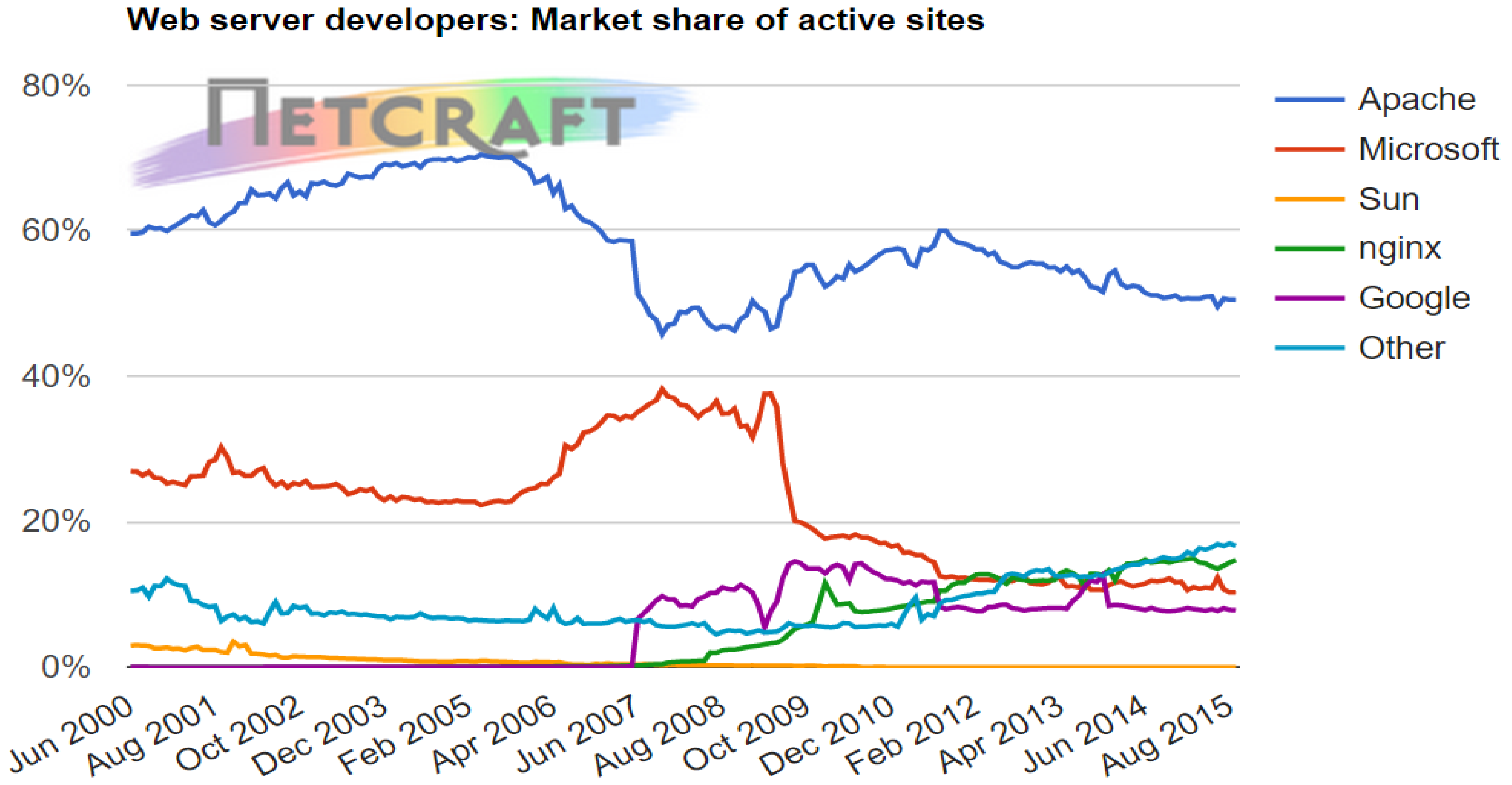
HTTP Servers

What are HTTP servers?

- It keeps listening for incoming connections at a specific port (default = 80)
- It processes HTTP requests and sends out replies in the form of HTTP responses
- It parses requests and sends the request to other handles if necessary (e.g. when dynamic content is required)

HTTP Servers

Common
Web server
applications



Ref: http://news.netcraft.com/archives/2015/09/16/september-2015-web-server-survey.html

HTTP Servers

A simple HTTP server’s pseudo-code

```
Open socket, listen at port 80
While true:
      Accept socket connection from client
      While read == true:
            Read request data
            Process request data
            Output response
      Close connection
```

HTTP Servers

```
Open socket, listen at port 80
While true:
      Accept socket connection from client
      While read == true:
            Read request data
            Process request data
            Output response
      Close connection
```

Blocking function (wait until a client connection comes in)

Can take considerable amount of time depending on the computation required

Meanwhile, the server cannot accept new connections

HTTP Servers

The previous simple server is an example of a single-threaded server

- For Web applications/services, it is usually more than serving static files from disk
- Execution of business logic, updating databases, writing logs are common actions
- A single-threaded HTTP server cannot handle many clients at the same time

HTTP Servers

Other approaches

- Create a new process to handle a new request
- Create a new thread to handle a new request
- Create a pool of workers (either processes or threads) in advance to handle new requests
- Event-driven

Nginx Web Server

A web server "with a strong focus on high concurrency, performance and low memory usage"

- A free and open source software developed by Igor Sysoev (a Russian software engineer)
- Use an event-driven (asynchronous) approach to hand HTTP requests
- Avoid waiting for blocking system calls (e.g. read from socket, read from file in memory or from disk)

Addition functions such as reverse proxy with caching, load balancing, and support other new protocols such as SPDY or WebSockets

Serving Dynamic Content

HTTP Servers vs. Application Servers

For running a Website with mostly static content, a Web server is sufficient. However, building an application or service involves more complex server-side logic, and very often you will need to generate content dynamically., You need an application server too.

Q: Why do we want to have HTTP servers and application servers instead of a single server-side program?

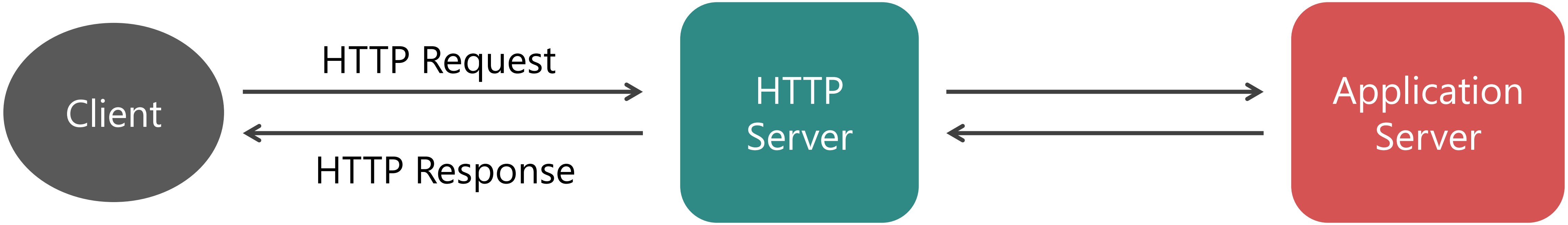
HTTP Servers vs. Application Servers

These two types of servers have different requirements

HTTP Servers	**Application Servers**
• Has to be stable and secure • Serve static files or content quickly • Be configurable • Be able to handle many requests at a time (concurrency issues) • Be language agnostic	• Execute business logic • Development using high-level languages is usually more efficient • Interface with other components to execute the business logic (e.g. database, message queues, other Web services)

Application Servers

The HTTP server will send requests to the application server for carrying out computation or for retrieving dynamic content

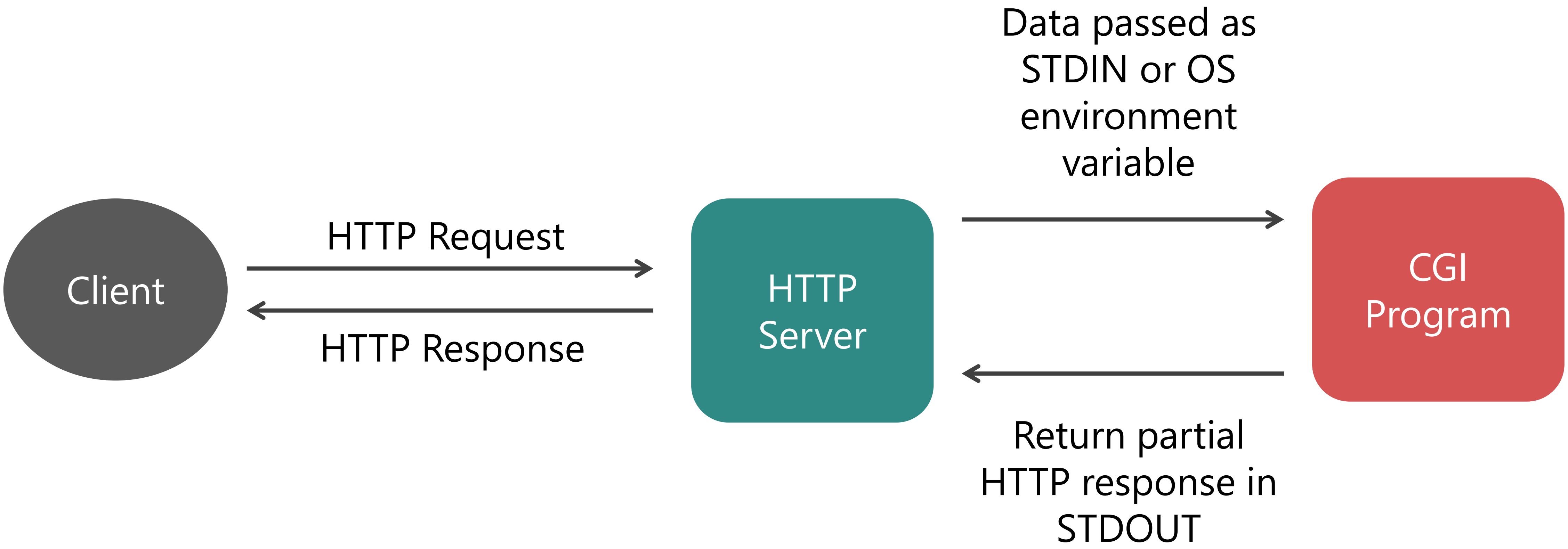


CGI

CGI = Common Gateway Interface

- A standard protocol for interfacing external application with a Web server
- CGI programs are executable programs that run on the Web server machine
- CGI programs in general would return HTML pages that are constructed dynamically
- Typical examples:
  - Visitor counter (Displaying the total number of visitors to a page)
  - Blog (retrieving the latest blog posts)

CGI



Client

HTTP Request

HTTP Response

HTTP Server

Data passed as STDIN or OS environment variable

Return partial HTTP response in STDOUT

CGI Program

CGI

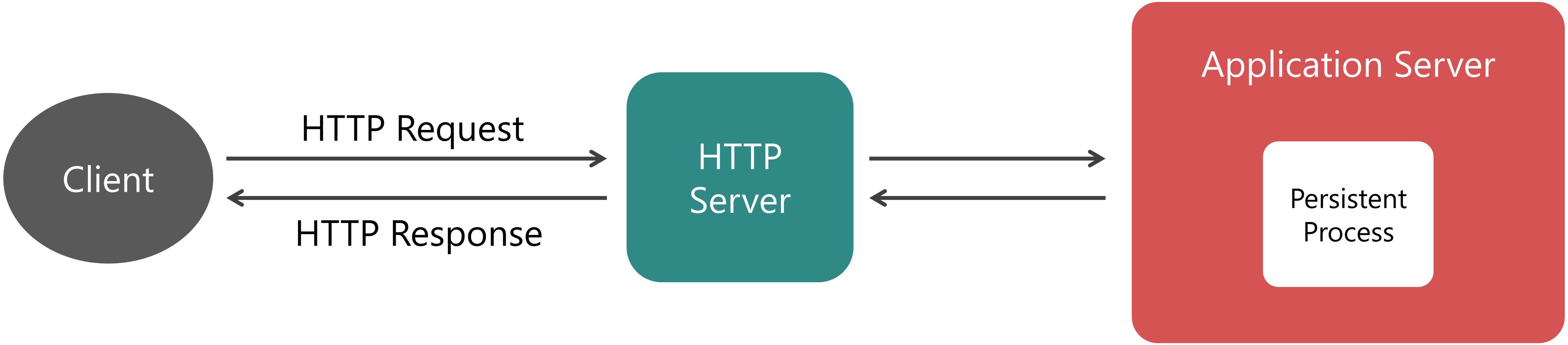
Limitation of CGI

- For each request to invoke a CGI program, a new process is created, which will be terminated at the end of the execution
- Thus, CGI is simple to implement, but is not efficient and not scalable
- The overhead to start and terminate the process can be huge (requires a lot of work in the OS when the workload is high)

E.g. imagine that the CGI program needs to load a huge dictionary from disk for performing translation of words

Other Methods

- In order to reduce the overhead to start and terminate a program, the program should be hosted in a server application, with a persistent process always running and ready to process requests.
- Some examples are mod_php or mod_python for Apache, FastCGI, SCGI, Python's WSGI



WSGI

WSGI refers to “Web Server Gateway Interface”

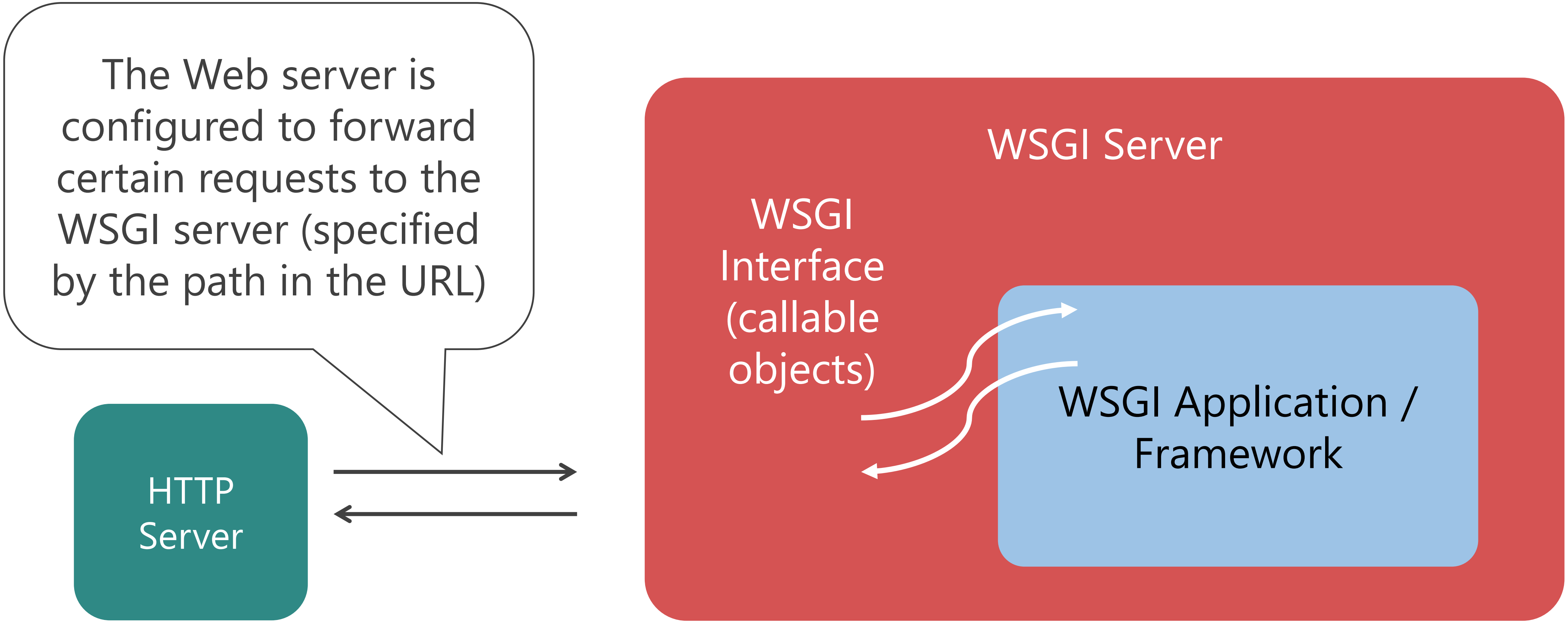
- Specify the interface through which a server and an application communicate
- If an application is written according to the specification, it will be able to run on any server developed according to the same specification
- Applications and servers that use the WSGI interface are said to be “WSGI compliant”
- WSGI applications can be stacked (more in the slides to come…)

WSGI

Why WSGI?

- Web servers are not capable of running Python applications
- For Apache, there is a module named mod_python, which enables Apache to execute Python codes
- However, mod_python is
  - not a standard specifications
  - no longer under active development
- Hence, the Python community came up with WSGI as a standard interface for Python Web applications

WSGI



The Web server is configured to forward certain requests to the WSGI server (specified by the path in the URL)

HTTP Server

WSGI Server

WSGI Interface (callable objects)

WSGI Application / Framework

Ref: http://www.fullstackpython.com/wsgi-servers.html

WSGI Servers vs. WSGI Applications

Why do we have WSGI servers and WSGI applications?

It is an example of de-coupling:

- **Applications** focus on how to get things done (e.g. business logic, updating databases, serving dynamic content, etc.)
- **Servers** focus on how to route requests, handle simultaneous connections, optimise computing resources, etc.
- As an application developer, you can focus on developing the functions and features, without worrying about how to interface with the Web server

Communication between the App and the Server

When a new request comes to the WSGI sever:

1. The server invokes the corresponding function in the application
2. Parameters are passed to the application using environment variables
3. The server also provides a callback function to the application
4. The application processes the request
5. The application returns the response to the server using the callback function provided by the server

Example

A simple WSGI-compatible application that returns “Hello World”

```
def application(environ, start_response):
      start_response(‘200 OK’, [(‘Content-Type’, ‘text/plain’)])
      yield ‘Hello World\n’
```

“environ” contains parameters that the server passes to the application (e.g. parameters in the query string)

“start_response” is a callback function provided by the server, the application uses it to return the HTTP response

Developing WSGI Applications

You do not need to directly implementing the WSGI interface in your application, as there are many “frameworks” that will help you development an application more easily

In this course, you are recommended to build one using the Flask framework (http://flask.pocoo.org/)

- Relatively easy to pick up
- Debug mode that assists your development
- Many plugins and modules

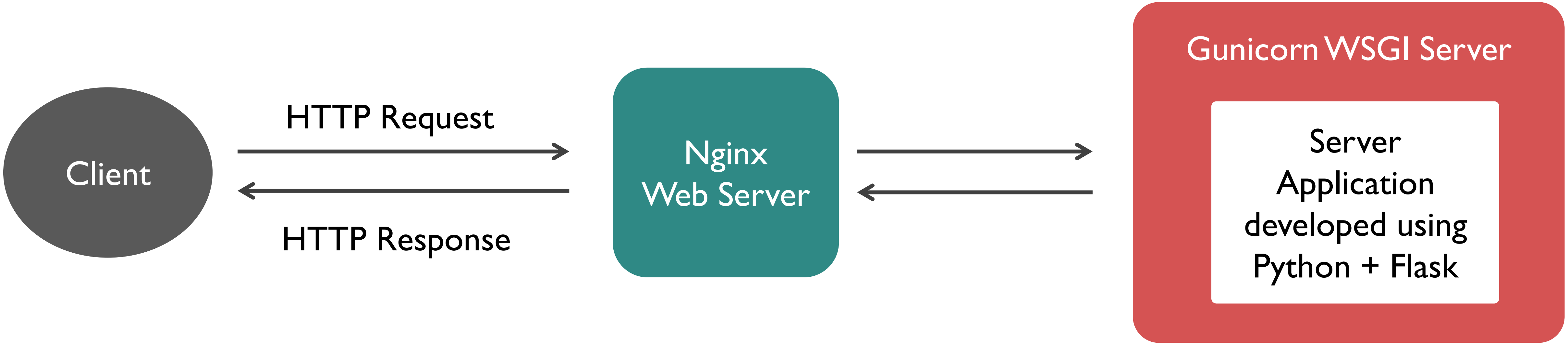
Developing WSGI Applications

Other options:

- Django (https://www.djangoproject.com/)
  A comprehensive Web framework following the model-view-controller (MVC) architectural pattern
- Bottle (http://bottlepy.org/)
  A micro-framework like Flask, but more lightweight and requires no dependencies on other modules

For more, see https://wiki.python.org/moin/WebFrameworks/

Web + Application Server



Python Application Server

In this course, you will build your server application using the following components

- Nginx as the HTTP server
- Gunicorn as a WSGI server
- Python as the programming language
- Flask as the Web framework

Developing Web Applications using Python & Flask

Python + Flask

Flask is a Python framework for developing WSGI compatible Web / HTTP applications

- http://flask.pocoo.org/
- In Ubuntu, install Flask using the following command

```
$ sudo pip install Flask
```

Flask ‘Hello World’

A Flask ‘Hello World’ Application

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

@app.route('/')
def hello_world():
    return 'Hello World!'

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

Define a Flask app

Define a route – binding a function to a URL of the server

Run the app (Using Flask’s internal Web server, not for deployment)

Flask Applications

How is this related to
Android app development?

Flask Applications

Suppose we are developing a simple news reading app with the following **functions**:

- Display the latest 50 news articles
- Users can ‘like’ an article

Both functions require the support of a **server**

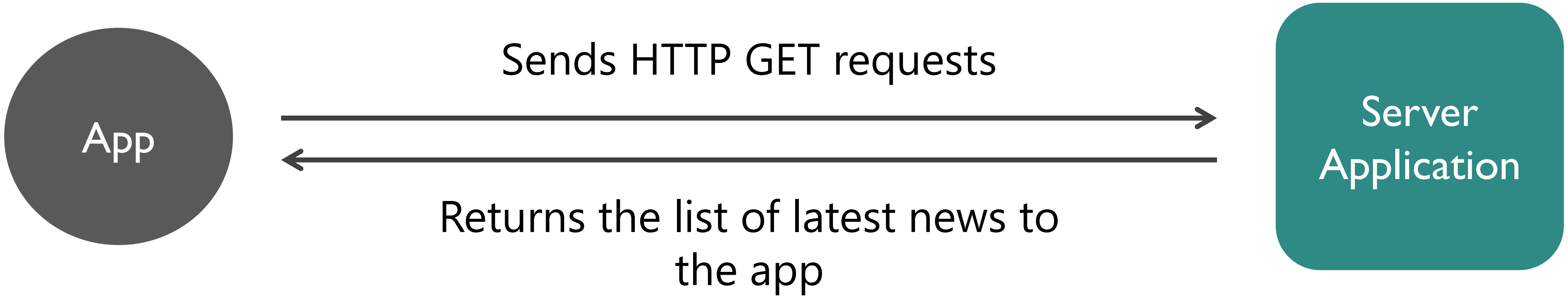
- Serves new articles when the app requests
- Updates the database when the user ‘likes’ an article

Flask Applications

We will need to develop a server application with at least the following two APIs (Application Programming Interfaces)

- Get latest news (GET)
- Like article (PUT or POST)

For example:



Flask Applications

A route in a Flask application can be considered as an API for the client to perform a particular function

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

@app.route('/')
def hello_world():
    return 'Hello World!'

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

This API returns the ‘Hello World!’ string

Flask Applications

More examples

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

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

Note: If you define the url with a trailing slash, like @app.route('/about/'), accessing it without a trailing slash will cause Flask to redirect to the canonical URL with the trailing slash. However, if you define the url without a trailing slash, like @app.route('/about'), accessing it with a trailing slash will produce a 404 “Not Found” error.

Flask Applications

Making certain parts of the URL dynamic

```
@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
```

The value given I the URL will be accessible in the function’s variable with the same name

The post_id must be an integer

Flask Applications

By default, a route only answers to HTTP GET requests, but you can change it by providing the methods explicitly when defining the route

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

Accessing Request Data

Your app will almost always need to pass some data to the server

- In the query string when using GET
  (e.g. the ID of a news article)
- In the HTTP body when using POST
  (e.g. the username and password for signing in)

You can access the data submitted from the client using the **request** object in Flask

Accessing Request Data

To access data in the query string (GET request)

```
@app.route('/get_news', methods=['GET'])
def get_news():
    news_id = request.args.get("news_id")
    ...
```

To access data in a POST request

```
@app.route('/like_news', methods=[POST'])
def like_news():
    news_id = request.form.get("news_id")
    ...
```

Returning JSON Data

When you are developing an API for a mobile app, usually you would like to return data in JSON format.

1. Using Python’s build –in JSON module

```
import json

@app.route('/get_news', methods=['GET'])
def get_news():
    news_id = request.args.get("news_id")
    articles = getNewsFromDatabase(news_id)
    output = json.dumps(articles)
    return output
```

Returning JSON Data

2. Using Flask’s build-in jsonify function

```
from flask import jsonify

@app.route('/get_news', methods=['GET'])
def get_news():
    news_id = request.args.get("news_id")
    articles = getNewsFromDatabase(news_id)
    return jsonify(status=“OK”, data=articles)
```

Example of an API

Let’s assume we are developing an API for finding the factorial of a number:

```
from flask import jsonify

@app.route('/add', methods=['GET'])
def add():
    a = request.args.get("a", 0, type=int)
    b = request.args.get("b", 0, type=int)
    sum = a + b
    return jsonify(status="OK", sum=sum)
```

```
GET: /add?a=5&b=6

{
    "status": "OK",
    "sum": 11
}
```

Testing a Flask Application

You can test your application by simply execute the python script, for example:

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

By default, Flask will execute the app using an internal server on port 5000, and the APIs can only be accessed from within the same machine (note the 127.0.0.1 address)

Make the server publicly available by using app.run(host='0.0.0.0')

Flask Debug Mode

For testing and debugging purposes, you can enable the DEBUG mode of Flask by

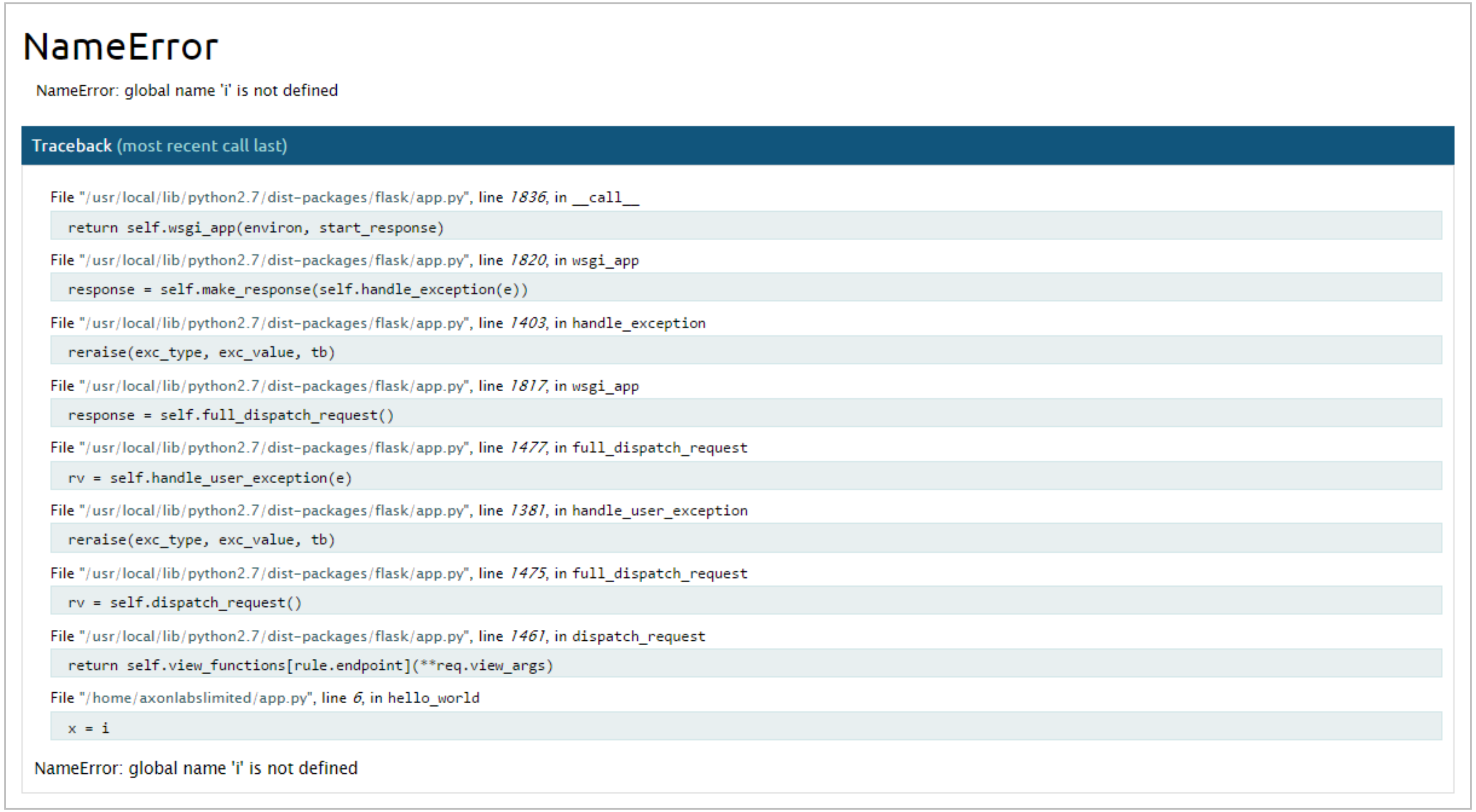
```
app.debug = True
app.run()
```

And then you will see:

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 211-226-346
```

Flask Debug Mode

If some problem happens, you will see a DEBUG interface when you access the URL. For example:



NameError

NameError: global name 'i' is not defined

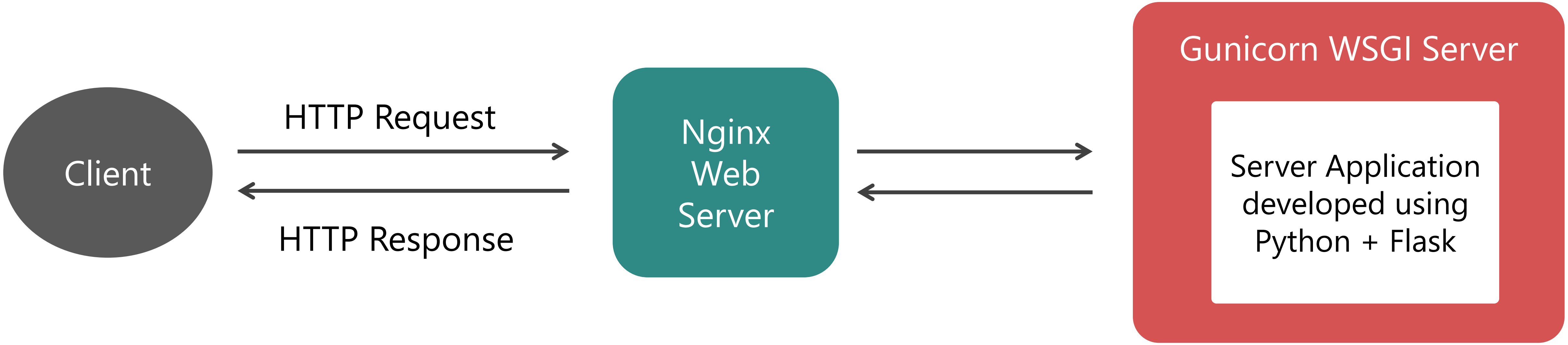
Traceback (most recent call last)

```
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1836, in __call__
  return self.wsgi_app(environ, start_response)
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1820, in wsgi_app
  response = self.make_response(self.handle_exception(e))
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1403, in handle_exception
  reraise(exc_type, exc_value, tb)
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1817, in wsgi_app
  response = self.full_dispatch_request()
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1477, in full_dispatch_request
  rv = self.handle_user_exception(e)
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1381, in handle_user_exception
  reraise(exc_type, exc_value, tb)
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1475, in full_dispatch_request
  rv = self.dispatch_request()
File "/usr/local/lib/python2.7/dist-packages/flask/app.py", line 1461, in dispatch_request
  return self.view_functions[rule.endpoint](**req.view_args)
File "/home/axonlabslimited/app.py", line 6, in hello_world
  x = i
```

NameError: global name 'i' is not defined

Deploying Flask Applications Using Gunicorn

System Architecture



Gunicorn

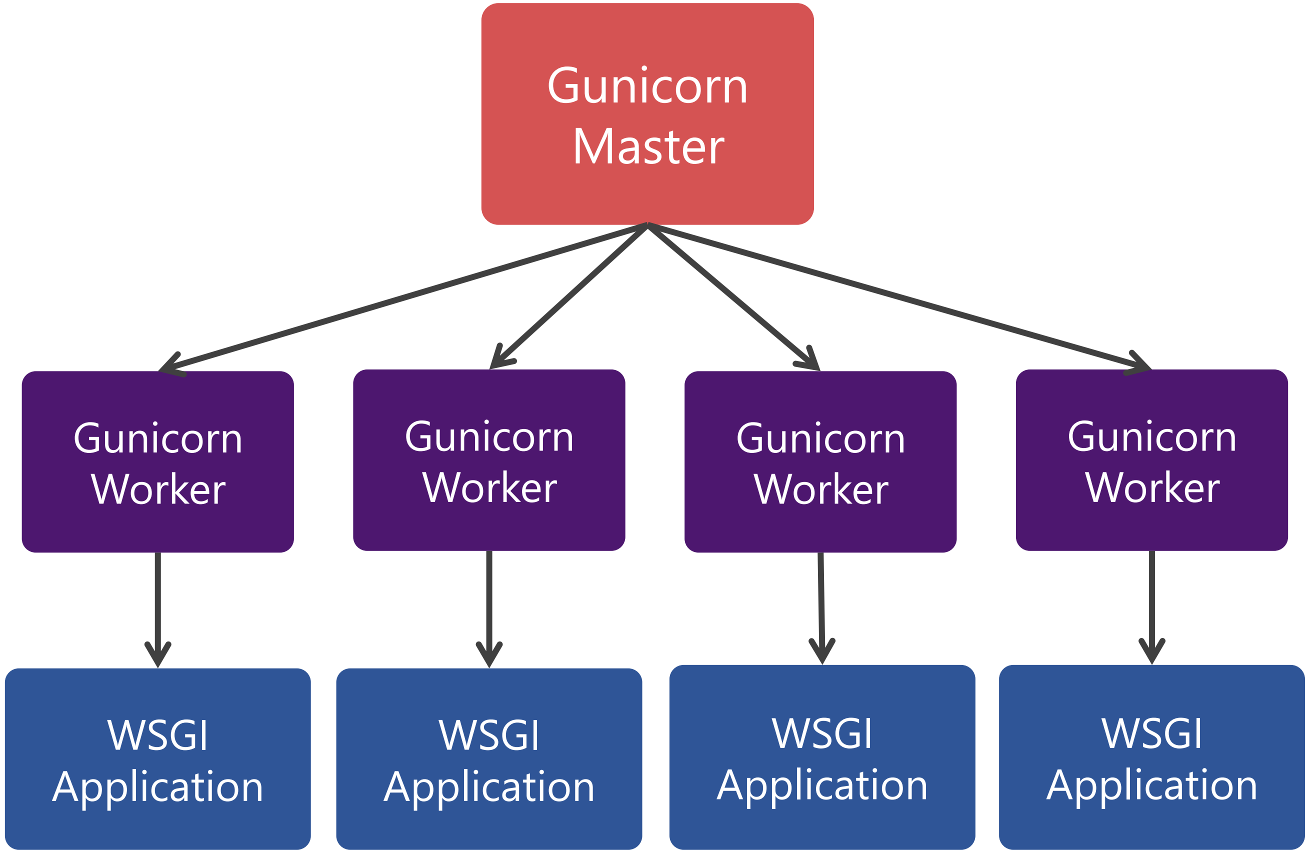
Gunicorn is a Python WSGI HTTP Server for Unix / Linux systems.

- It acts as a container of a WSGI application
- It manages one or more instances of the application (multiple workers)
- Official Website: http://gunicorn.org/
- Documentation: http://docs.gunicorn.org/

Gunicorn

Architecture of Gunicorn

- A pre-fork worker model
- A master process managers a set of worker processes
- Each worker process runs a copy of your application



```
$ gunicorn app:app -b localhost:8000 -w 4
```

Gunicorn

- **Basic Usage**

```
$ gunicorn [OPTIONS] $(MODULE_NAME):$(VARIABLE_NAME)
```

- **Example**:
  - A Flask app called ‘app’ defined inside a file called app.py
  - Running on port 8000 on localhost
  - Create 4 worker processes

```
$ gunicorn app:app -b localhost:8000 -w 4
```

Gunicorn Workers

How to determine the suitable number of workers?

- Depends on your application's design and also the configurations of the server (e.g. number of cores of CPUs)
- In general: 2n + 1 (n = number of cores)
- Based on the assumption that half of the workers are doing I/O while half of the workers are doing computation

Gunicorn Workers

There are TWO main types of Gunicorn workers

1. **Sync Workers**
   - Default type – handles a single request at a time
   - Suitable for applications that do not do something that consume an undefined amount of time or resources
2. **Async Workers**
   - For non-blocking request processing
   - Use this if your application has I/O bound operations (i.e. need to wait for I/O events to finish)
   - However, failure in a process might affect many requests

Gunicorn Workers

Consider an example:

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

@app.route(“/sleep/”)
def go_sleep():
    x = random.randint(1,3)
    time.sleep(x)
    return str(x)

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

If you use sync workers, a worker can only serve a new request after one request has been finished

Using async workers (e.g. gevent or eventlet), a worker will switch to serve another request while one is waiting for I/O (or any other blocking operation)

Using Nginx as a Reverse Proxy and Load Balancer

Nginx as a Reverse Proxy

Nginx is a Web server but can also be configured as a reverse proxy server

- It can proxy requests to another HTTP server or a non-HTTP server
- It supports the following non-HTTP protocol:
  FastCGI, uwsgi, SCGI, memcached
- It can buffer responses from servers to improve performance
  (when the client is slow)

Configuring Nginx

Nginx can be configured by editing the configuration files

- In Ubuntu, configuration files are usually stored under /etc/nginx/
- A main configuration file named “nginx.conf”
- One or more configuration files for each of the sites hosted by the server
  (see “/etc/nginx/site-available” and “/etc/nginx/site-enabled”)

Ref: http://nginx.org/en/docs/http/load_balancing.html

Configuring Nginx

Basic structure of a configuration file for Nginx

```
user www-data;
worker_processes 4;
pid /run/nginx.pid;

events {
    ...
}

http {
    server {
        location {
            ...
        }
    }
    ...
}
```

The “main” context

The “event” context

The “http”, “server” and “location” contexts

Configuring Nginx

A simple setup for a Web server serving content from a particular directory

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    location / {
        root /data/www;
    }

}
```

Assume that the server is up at http://www.myserver.com/, then a request to http://www.myserver.com/images/x.jpg will retrieve an image named “x.jpg” from the directory /data/www/images

Configuring Nginx

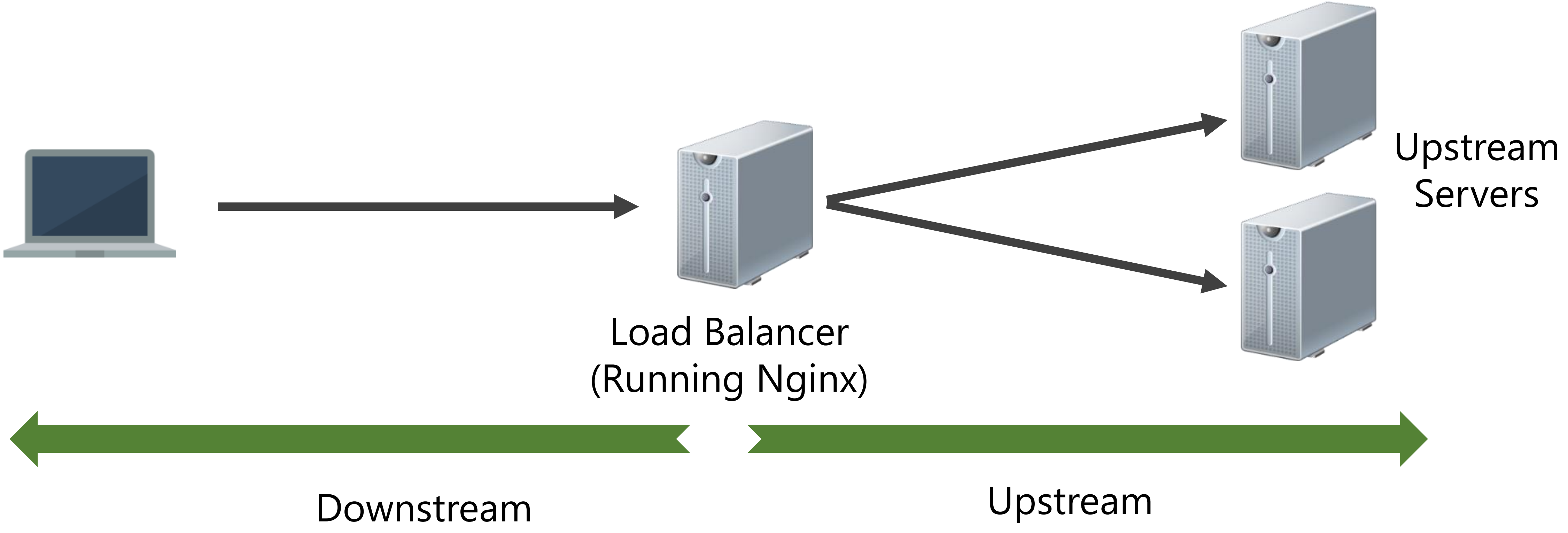
Serving files from different directories

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    location / {
        root /data/www;
    }

    location /images/ {
        root /data/images;
    }

}
```

Configuring Nginx

Configuring Nginx as a reverse proxy load balancer is simple.

However, before that we introduce the term “upstream”



Configuring Nginx

Load balancing example in Nginx

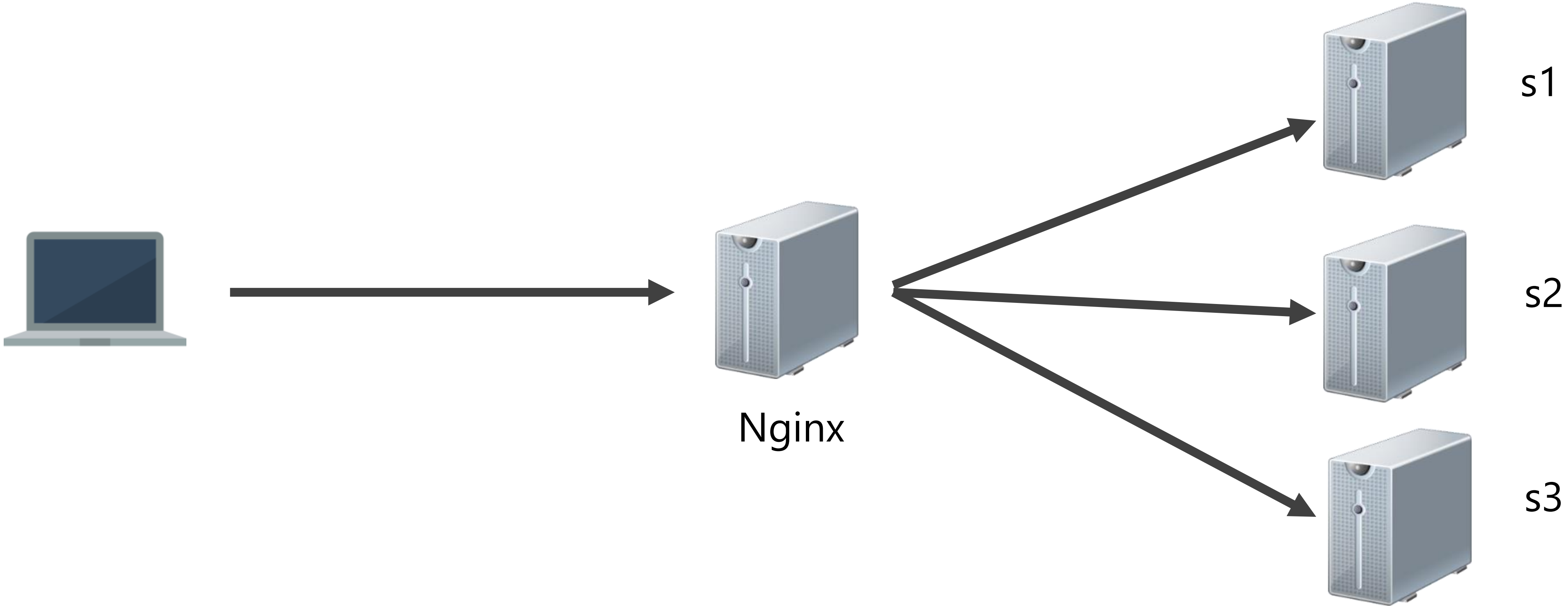
Note:

When no load balancing method is specified, the round-robin method will be used

```
http {
    upstream myservers {
        server s1.myserver.com;
        server s2.myserver.com;
        server s3.myserver.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myservers;
        }
    }
}
```

Configuring Nginx

The example in the previous slide is illustrated in the following diagram:



Configuring Nginx

Load balancing using the “least connection” method

```
http {
    upstream myservers {
        least_conn;
        server s1.myserver.com;
        server s2.myserver.com;
        server s3.myserver.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myservers;
        }
    }
}
```

Configuring Nginx

Persistence (Stickiness)

- Sometimes we need the same server to serve the same client for a series of requests (why?)
- Round-robin and least connected methods do NOT guarantee that the same client will be served by the same server
- Persistence (or stickiness) refers to the ability of the load balancing to forward requests to the same server

Configuring Nginx

Use IP hashing as the load balancing method to achieve persistence in Nginx

```
http {
    upstream myservers {
        ip_hash;
        server s1.myserver.com;
        server s2.myserver.com;
        server s3.myserver.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myservers;
        }
    }
}
```

Configuring Nginx

Other functions include:

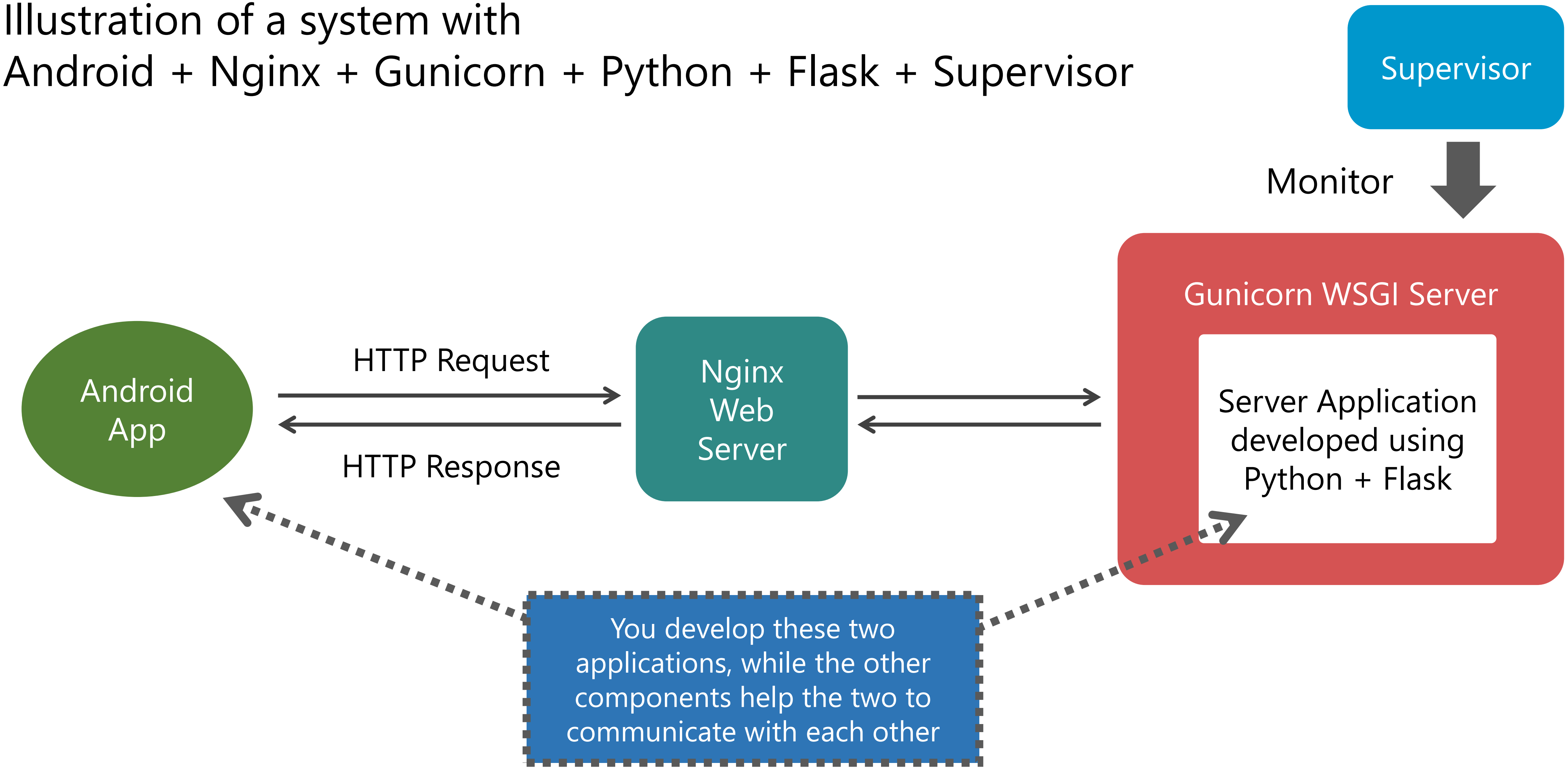
- Health checks of servers
- Buffering server response
- Routing requests to applications (e.g. to a Python Web app)

Ref: https://www.nginx.com/resources/admin-guide/load-balancer/

Summary

Summary

Illustration of a system with
Android + Nginx + Gunicorn + Python + Flask + Supervisor



You develop these two applications, while the other components help the two to communicate with each other

Next Lecture:
Databases and Caches

End of Lecture 5