Writing a WSGI Web Framework from Scratch

Mohammad "Kiyarash" Fazeli

Maktabkhooneh.org

October 9, 2024

Workshop Outline

Introduction and Objectives

- Understand the evolution of Python web application deployment.

Introduction and Objectives

- Understand the evolution of Python web application deployment.
- Learn the basics of WSGI and its importance.

Introduction and Objectives

- Understand the evolution of Python web application deployment.
- Learn the basics of WSGI and its importance.
- Build a simple WSGI application.

Introduction and Objectives

- Understand the evolution of Python web application deployment.
- Learn the basics of WSGI and its importance.
- Build a simple WSGI application.
- Explore libraries that simplify development.

Introduction and Objectives

- Understand the evolution of Python web application deployment.
- Learn the basics of WSGI and its importance.
- Build a simple WSGI application.
- Explore libraries that simplify development.
- Examine popular frameworks' WSGI implementations.

Historical Methods of Running Web Code: CGI

CGI (Common Gateway Interface)

- Separate process per request.

Historical Methods of Running Web Code: CGI

CGI (Common Gateway Interface)

- Separate process per request.
- High overhead, poor scalability.

Historical Methods of Running Web Code: CGI

CGI (Common Gateway Interface)

- Separate process per request.
- High overhead, poor scalability.
- Code example

Historical Methods of Running Web Code: mod_python

mod_python

- Apache module for Python.

FastCGI

- Persistent processes.
- Improved performance over CGI.

Historical Methods of Running Web Code: mod_python

mod_python

- Apache module for Python.
- Better performance but Apache-specific.

FastCGI

- Persistent processes.
- Improved performance over CGI.

Historical Methods of Running Web Code: mod_python

mod_python

- Apache module for Python.
- Better performance but Apache-specific.
- code

FastCGI

- Persistent processes.
- Improved performance over CGI.

Historical Methods of Running Web Code: FastCGI

FastCGI

- Persistent processes.

Historical Methods of Running Web Code: FastCGI

FastCGI

- Persistent processes.
- Improved performance over CGI.

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes
- **Communication Mechanism**:

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes
- **Communication Mechanism**:
  - **CGI**: Environment variables and I/O

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes
- **Communication Mechanism**:
  - **CGI**: Environment variables and I/O
  - **FastCGI**: Efficient binary protocol

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes
- **Communication Mechanism**:
  - **CGI**: Environment variables and I/O
  - **FastCGI**: Efficient binary protocol
- **Concurrency**:

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes
- **Communication Mechanism**:
  - **CGI**: Environment variables and I/O
  - **FastCGI**: Efficient binary protocol
- **Concurrency**:
  - **CGI**: Sequential, one-at-a-time

FastCGI vs. CGI: Architectural and Performance Differences

- **Process Lifetime**:
  - **CGI**: Separate processes for each request
  - **FastCGI**: Long-lived, persistent processes
- **Communication Mechanism**:
  - **CGI**: Environment variables and I/O
  - **FastCGI**: Efficient binary protocol
- **Concurrency**:
  - **CGI**: Sequential, one-at-a-time
  - **FastCGI**: Concurrent request handling

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:
  - **CGI**: Slow process creation, limited resources

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:
  - **CGI**: Slow process creation, limited resources
  - **FastCGI**: Faster request handling, efficient resource utilization

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:
  - **CGI**: Slow process creation, limited resources
  - **FastCGI**: Faster request handling, efficient resource utilization
- **Scalability**:

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:
  - **CGI**: Slow process creation, limited resources
  - **FastCGI**: Faster request handling, efficient resource utilization
- **Scalability**:
  - Process Creation Overhead

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:
  - **CGI**: Slow process creation, limited resources
  - **FastCGI**: Faster request handling, efficient resource utilization
- **Scalability**:
  - Process Creation Overhead
  - Resource Utilization

FastCGI vs. CGI: Architectural and Performance Differences

- **Performance**:
  - **CGI**: Slow process creation, limited resources
  - **FastCGI**: Faster request handling, efficient resource utilization
- **Scalability**:
  - Process Creation Overhead
  - Resource Utilization
  - Scaling Challenges

Need for Standardization

- Fragmentation in Python web development.
- Incompatibilities between servers and applications.
- Introduction of WSGI to provide a standard interface.

Case Study Overview

- A web application facing scalability issues.
- Limitations with Django and Apache.
- High number of concurrent connections.

Challenges Faced

- **Django Limitations**
  - Overhead not suitable for simple applications.
  - Difficult to optimize for specific needs.
- **Apache Limitations**
  - Process/thread per connection.
  - Resource-intensive under high load.

Solution: Custom WSGI Framework

- Built a lightweight framework tailored to the application's needs.
- Improved performance and scalability.
- Greater control over resource management.

What is WSGI?

- **Web Server Gateway Interface**
- A standard interface between web servers and Python web applications.
- Defined in PEP 3333.

WSGI Components

- **Application Callable**
- **`environ` Dictionary**
- **`start_response` Callable**

Benefits of WSGI

- Promotes interoperability between frameworks and servers.
- Simplifies deployment and scaling.
- Encourages the development of middleware and reusable components.

Hello World WSGI Application

Code Example:

```
def application(environ, start_response):
    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=u
    start_response(status, headers)
    return [b"Hello, World!"]
```

Explanation of Components

- `environ`: Contains request data.
- `start_response`: Starts the HTTP response.
- **Return Value**: An iterable yielding the response body.

Framework Structure

- Organize code for scalability.
- Separate concerns: routing, handling requests, generating responses.

Implementing URL Routing

Example Route Mapping:

```
routes = {
    '/': home_view,
    '/about': about_view,
}
```

- Map URLs to view functions.
- Handle dynamic URLs with parameters.

Handling Requests and Responses

Manual Parsing:

- Extract query parameters from `environ`.
- Build response headers and body.

Limitations of Pure Python Implementation

- Complexity in parsing and handling data.
- Potential security risks.
- Reinventing the wheel.

Using WebOb

Code Example:

```
from webob import Request, Response

def application(environ, start_response):
    request = Request(environ)
    response = Response()
    response.text = "Hello, World!"
    return response(environ, start_response)
```

Using Werkzeug

Code Example:

```
from werkzeug.wrappers import Request, Response

@Request.application
def application(request):
    return Response('Hello, World!')
```

Benefits of Using Libraries

- Simplify request and response handling.
- Provide robust, tested components.
- Save development time and reduce errors.

Django's WSGI Implementation

- Uses `wsgi.py` file.
- `get_wsgi_application()` function sets up the application.

Flask's WSGI Integration

- The Flask app object is a WSGI application.
- Can access the underlying WSGI application via `app.wsgi_app`.

Bottle's WSGI Approach

- The default Bottle app is a WSGI application.
- Simple and lightweight, ideal for small applications.

What is ASGI?

- **Asynchronous Server Gateway Interface**
- Designed for asynchronous Python web applications.
- Supports long-lived connections like WebSockets.

Why ASGI?

- Modern web applications require asynchronous capabilities.
- WSGI is synchronous and cannot handle async code efficiently.
- ASGI enables high-performance async frameworks like FastAPI.

Recap

- Explored the evolution of Python web deployment.
- Built a simple WSGI application and framework.
- Introduced libraries to simplify development.
- Examined popular frameworks' WSGI implementations.
- Briefly discussed ASGI and asynchronous programming.

Additional Resources

- PEP 3333: WSGI Specification
- ASGI Documentation
- Werkzeug Documentation
- WebOb Documentation

Questions?

Thank you for your attention!

Feel free to ask any questions.

Contact Information

- **Email**: your.email@example.com
- **GitHub**: github.com/yourusername
- **LinkedIn**: linkedin.com/in/yourprofile