Writing a WSGI Web Framework from Scratch

Mohammad "Kiyarash" Fazeli

Maktabkhooneh.org

October 10, 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.

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.

Benefits of Using Libraries

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

Introduction to Shortly

- Build by the GOAT
- Goal: Build a URL shortener using Werkzeug
- Werkzeug: Utility library for WSGI applications
- Approach: Create flexible web applications

WSGI Basics

```python
from werkzeug.wrappers import Request, Response

def application(environ, start_response):
    request = Request(environ)
    text = f'Hello, {request.args.get("name", "World")}!'
    response = Response(text, mimetype='text/plain')
    return response(environ, start_response)
```

Creating the Application

- Set up ‘shortly.py‘
- Class-based application with WSGI support

```
class Shortly:
    def wsgi_app(self, environ, start_response):
        request = Request(environ)
        response = self.dispatch_request(request)
        return response(environ, start_response)

    def dispatch_request(self, request):
        return Response('Hello, World!')
```

Shortly Class Skeleton

```
class Shortly(object):
    def __init__(self, config):
        self.redis = redis.Redis(
            config['redis_host'], config['redis_port'], decode_resp
        )

    def dispatch_request(self, request):
        return Response('Hello World!')

    def wsgi_app(self, environ, start_response):
        request = Request(environ)
        response = self.dispatch_request(request)
        return response(environ, start_response)

    def __call__(self, environ, start_response):
        return self.wsgi_app(environ, start_response)
```

Setting Up the Environment

- Initialize Jinja2 environment
- Define `render_template` method

Template Rendering

```
def __init__(self, config):
    # ...
    template_path = os.path.join(os.path.dirname(__file__), 'templa
    self.jinja_env = Environment(loader=FileSystemLoader(template_pa
                                 autoescape=True)

def render_template(self, template_name, **context):
    t = self.jinja_env.get_template(template_name)
    return Response(t.render(context), mimetype='text/html')
```

Routing

- Define URL routes using `Map` and `Rule`
- Routes:
  - `'/'` → `new_url`
  - `'/<short_id>'` → `follow_short_link`
  - `'/<short_id>+'` → `short_link_details`
- Implement `dispatch_request` method

Dispatch Request Method

```
def dispatch_request(self, request):
    adapter = self.url_map.bind_to_environ(request.environ)
    try:
        endpoint, values = adapter.match()
        return getattr(self, f'on_{endpoint}')(request, **values)
    except HTTPException as e:
        return e
```

First View: `on_new_url`

- Handle URL submission and validation
- Render template or redirect

on_new_url Method

```
def on_new_url(self, request):
    error = None
    url = ''
    if request.method == 'POST':
        url = request.form['url']
        if not is_valid_url(url):
            error = 'Please enter a valid URL'
        else:
            short_id = self.insert_url(url)
            return redirect(f"/{short_id}+")
    return self.render_template('new_url.html', error=error, url=ur
```

URL Validation

- Define `is_valid_url` function
- Check scheme is `http` or `https`

is_valid_url Function

```
def is_valid_url(url):
    parts = url_parse(url)
    return parts.scheme in ('http', 'https')
```

Inserting URLs into Redis

- Check for existing short ID
- Generate new short ID if necessary
- Store URL and reverse lookup

insert_url Method

```
def insert_url(self, url):
    short_id = self.redis.get(f'reverse-url:{url}')
    if short_id is not None:
        return short_id
    url_num = self.redis.incr('last-url-id')
    short_id = base36_encode(url_num)
    self.redis.set(f'url-target:{short_id}', url)
    self.redis.set(f'reverse-url:{url}', short_id)
    return short_id
```

Generating Short IDs

- Convert incremented number to base36
- Use custom `base36_encode` function

base36_encode Function

```
def base36_encode(number):
    assert number >= 0, 'positive integer required'
    if number == 0:
        return '0'
    base36 = []
    while number != 0:
        number, i = divmod(number, 36)
        base36.append('0123456789abcdefghijklmnopqrstuvwxyz'[i])
    return ''.join(reversed(base36))
```

Redirect View

- `on_follow_short_link` method
- Retrieve target URL from Redis
- Increment click count

on_follow_short_link Method

```
def on_follow_short_link(self, request, short_id):
    link_target = self.redis.get(f'url-target:{short_id}')
    if link_target is None:
        raise NotFound()
    self.redis.incr(f'click-count:{short_id}')
    return redirect(link_target)
```

Detail View

- `on_short_link_details` method
- Display link target and click count
- Handle missing URLs

on_short_link_details Method

```
def on_short_link_details(self, request, short_id):
    link_target = self.redis.get(f'url-target:{short_id}')
    if link_target is None:
        raise NotFound()
    click_count = int(self.redis.get(f'click-count:{short_id}') or 0)
    return self.render_template('short_link_details.html',
        link_target=link_target,
        short_id=short_id,
        click_count=click_count
    )
```

Templates

- Use Jinja2 for templating
- Create `layout.html` as base template
- Extend base template in other templates

Template Files

- `layout.html`: Base structure
- `new_url.html`: URL submission form
- `short_link_details.html`: Link details

Styling

- Add CSS styles in `static/style.css`
- Basic styling for layout and elements

Testing the Application

- Run the server
- Submit new URLs
- Access short links
- View link details and click counts

Bonus: Refinements

- Implement custom 404 page
- Explore additional features
- Reference example in Werkzeug repository

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