# 3D06: Software Design Analysis

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

## Group 16

Chosen Project: Flask

Flask
web development, one drop at a time

| Team Members | Student ID |
|---|---|
| Lakshaya Sabharwal | 23364363 |
| Suhani Singla | 23364401 |
| Niall Grogan | 21364215 |
| Raaghav Sawhney | 23364370 |

## Contents

## Introduction:

Flask is a simple effective web framework for Python known for its user-friendliness. It helps developers quickly create web applications with less code. Flask acts as a bridge between web servers and applications, using the WSGI standard to manage this interaction. (1)

It incorporates Werkzeug for handling requests and routing, (2), along with Jinja2 for rendering web page templates (3).

Flask's design can be expanded with various plugins that add capabilities like database integration and user authentication. These help developers as they can add more specialized components as when they need them. Flask also has strong documentation and a supportive community across the globe.

We wanted to learn more about who its stakeholders are and how they affect the software. The group were also interested in other views of the application, including development, deployment, operational, context, and information views. We also wanted to understand how the application's code was written and how efficient it was in running.

After creating various views, software metrics were calculated, Flask was found to have a high maintainability index and an average response time of 0.43 seconds. An interesting finding was that the Flask uses ~38 MiB of memory, while its competitor Django required 60 MiB of memory for similar tasks. The Unit Tests found an average code coverage of 94%, with certain tests being skipped and others having very low coverage rates.

Our contribution consisted of 4 aspects. In the first, we provide documentation on the operation of Flask that we believed was not explained thoroughly in the documentation. Test Case Coverages were also improved upon, resulting in an overall increase of 3%, as well as non-executing & skipped tests being executed. The questions from the project specification were also addressed.

*Why we chose this Project:*
We took on this task because we thought it provided a valuable opportunity to deepen our understanding of Python through practical analysis of an existing codebase using a variety of automated tests.

This "learning through analysing" approach enabled us to quickly learn coding practices and implement various code functionalities. Furthermore, the incomplete test coverage provided an opportunity for us to make meaningful contributions to this project.

The group also felt that engaging with a project with a diverse development team and an active community, such as Flask, would give us insights into real-world applications and dynamic team interactions, which would improve our learning experience

## Stakeholders:

Since Flask is a relatively small library, its stakeholders can be broadly categorized into the following. These were found using the approach described by Rozanski & Woods, (4).

*Acquirers:*
Flaks is a lightweight Python web development framework which is open sourced but is controlled by the pallet's projects organization

*Assessors:*
Flask relies on its community and contributors to ensure its legal compliances. It operates under the BSD license, (5). This grants the users freedom to use, modify and distribute under certain conditions.

Broadly speaking, its legal compliance also comes under the PSF (Python software Foundation) community, (6).

*Developers, Maintainers, Testers:*
Flask was developed by a group of international computer scientists in 2010 under the name POCOO (later the pallets organization), (7). Its founders are Armin Ronacher and Chris Jennings. While Ronacher still contributes from time-to-time in daily contributions much of the heavy lifting is carried out by the community of developers which take care of issues and testing.

Mainly, the pull requests are accepted by pallets team engineers with the tags @davidism, @greyli, (8). (Real name: David lord)

*Suppliers:*
Flask requires web-hosting services. Therefore, it relies on providers like AWS etc.

It also requires version control systems such as GitHub and GitLab. It also needs the PyPI (Python package index) because they are crucial for flask developers to manage dependencies, distribute flask extensions and related packages.

GitHub also helps in automating the tests.

*Users:*
There are a variety of organizations which use the flask framework for web development such as LinkedIn, Pinterest, Uber etc. What makes flask such a genius tool is that it can be easily be utilized by a single user to build a website on their own without much hassle.

A table has been to get a clearer view on this subject:

| CATEGORY | ENTITY |
|---|---|
| ACQUIRER | |
| ASSESSORS | python SOFTWARE FOUNDATION |
| DEVELOPERS, MAINTAINERS, TESTERS | ARMIN RONACHER, GitHub Community DAVID LORD |
| SUPPLIERS | aws, GitHub, PyPi |
| USERS | in, Pinterest, Web Developer |

Figure 1: Flask Stakeholders Diagram

**Views:**

<u>The Context View:</u>

*Definition:*

The context view focuses on the relationships, dependencies and interactions between the system (in this case, a Flask application) and its environment, (4).

*Key Modules:*

The main modules, most relevant to context view are:

*ctx.py:* This file directly addresses Flask's application and request contexts. In Flask, context is a temporary environment that holds information about the current request or application. This is how Flask manages several requests and applications simultaneously. Flask is tied to the interactions between the Flask application and its external environment, including HTTP requests and responses because it can track which application and request it is handling by switching these contexts in and out.

*globals.py:* This file defines proxies that allow the current application and request objects to be used anywhere in the code without needing to pass these objects as arguments. This is essential for understanding how Flask apps interact with their environment, as request and current_app play a key role in processing incoming requests and sending responses. These globals change based on the current thread handling the request, making it easier for your application to interact with the external environment correctly and securely.

*app.py:* This file is essential in setting up a Flask application, registering routes, and handling requests. It's where the Flask application starts and interacts with its environment by sending and receiving HTTP requests. The globals from globals.py and the contexts described in ctx.py are used by the functions and classes in app.py to interact with the application's environment.

These files are essential to understanding the context viewpoint because they specify how the Flask application interacts with the external environment (web servers, HTTP clients, databases, etc.) and builds environments (contexts) for handling incoming requests and maintain application data.

*UML Diagram:*

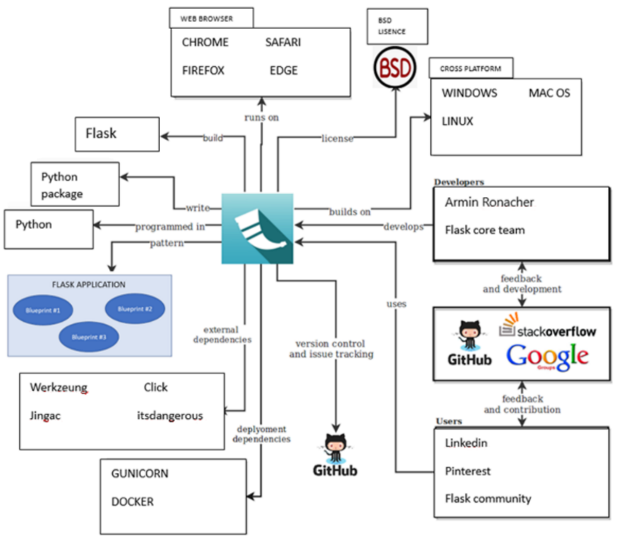


Figure 2: Flask Context View UML Diagram

<u>The Development View:</u>

The development view of a system details the areas of code that are likely most important to contributors who are involved in contributing to the code, (9). These contributors are likely engineers, developers or testers. By dividing the organisation of a codebase into a development view, it is much easier for the aforementioned contributors to understand its structure, dependencies, configurations etc. This leads to an improvement in productivity, maintainability and collaboration.

Some of the key modules identified in flask were as follows:

*"src/flask/app.py"* – This module contains the main Flask class "Flask", which is behind all flask applications. This file contains the definitions of many functions e.g. Error Handling Functions "raise_routing_exception", Initialising and Adding routes "__init__", Creating Test Clients "test_client".

*"src/flask/blueprintp.py"* – The purpose of blueprints is to allow developers to define common routes, essentially encapsulating these parts of an application into smaller units. This code file is key for maintaining the organisation of the file structure in the application files.

*"src/flask/templating.py"* – This module uses the Jinja templating engine, also maintained by Pallets (10), to define functions used to render templates. Templates are used to contain variables, which can be replaced by dynamic data requested by clients, when the template is rendered, (11). Templates can be reused, allowing developers to define common layouts easier.

*"src/flask/views.py"* – At the core of Flask applications are view functions. These are written to respond to requests made by the end application user, (12). The views handle the incoming URL request by matching it to the view that should look after it.

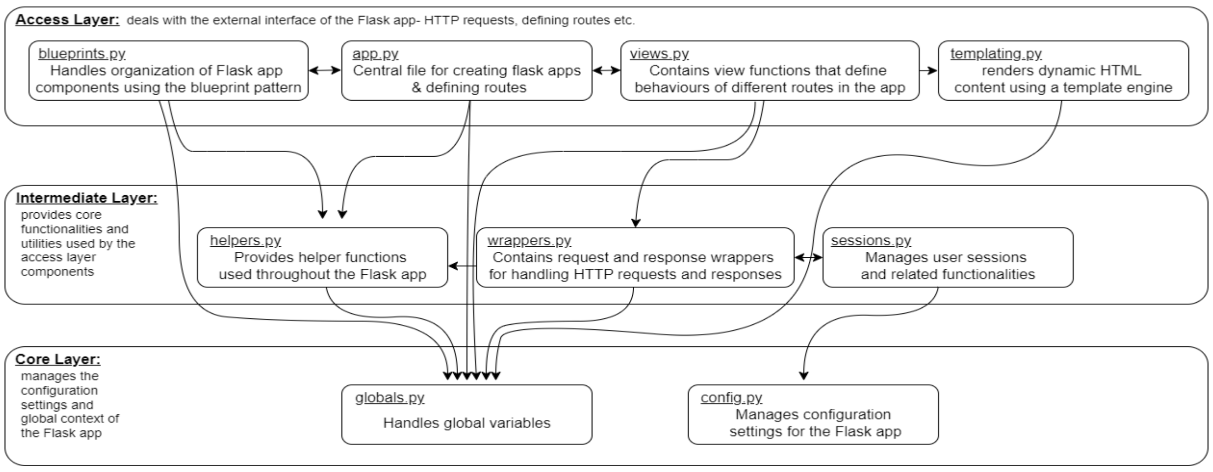

Access Layer: deals with the external interface of the Flask app- HTTP requests, defining routes etc.
blueprints.py Handles organization of Flask app components using the blueprint pattern
app.py Central file for creating flask apps & defining routes
views.py Contains view functions that define behaviours of different routes in the app
templating.py renders dynamic HTML content using a template engine
Intermediate Layer: provides core functionalities and utilities used by the access layer components
helpers.py Provides helper functions used throughout the Flask app
wrappers.py Contains request and response wrappers for handling HTTP requests and responses
sessions.py Manages user sessions and related functionalities
Core Layer: manages the configuration settings and global context of the Flask app
globals.py Handles global variables
config.py Manages configuration settings for the Flask app

Figure 3: Layered Architecture Development View

*Dependencies*
*Werkzeug* – WSGI Library (Web Server Gateway Interface Library) – Flask builds on its request and response handling, routing and for handling low-level codes.

*Jinja* – Templating Engine Discussed above – Flask uses it for creating placeholders for dynamic data and therefore for creating dynamic webpages.

*itsDangerous* – Secure cookies – Flask uses itsdangerous to implement user authentication etc.

*click* – Command Line Interfaces – Used by flask to manage cli tasks

*MarkupSafe* – Helps prevent Cross-site scripting attacks (XSS) – Used to maintain the security of Flask Apps

## Deployment View

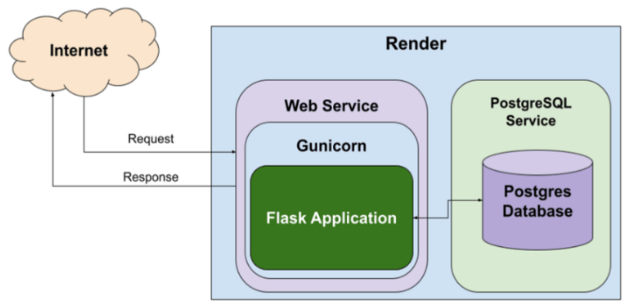


Figure 4: Deployment View Diagram

After developing for the web application is complete, it must be deployed. Deployment can be carried out either locally or online, where everyone can see.

If decided to deploy locally, the built-in server, debugger and reloader will be used.

Instead, when deploying the app online a dedicated WSGI Server or a hosting platform should be used, this way both Windows and Linux are supported.

It is also important that, the server used for testing development should not be used for deploying the application to millions of people and set up online, as this server intended to be used only for local changes and hence isn't stable or efficient.

Once the application is deployed, essential measures are also taken to ensure that the security key is changed to what was that at the time it was being developed.

A wheel file is also used. This a built-in package in python, the wheel files denoted by .whl are more consistent and efficient, hence are preferred. It has all the files for deployment and hence prevents errors if done manually.

There are self-hosted options as well as hosted platforms, examples of both are given below:

**Self-Hosted:** Gunicorn , Waitress, ASGI, mod_wsgi

**Hosted Platforms:** PythonAnywhere, GoogleAppEngine , GoogleCloudRun , MicrosoftAzure

If network dependencies are concerned, the Domain and DNS should be properly included so it correctly points to the server hosting application. Also, firewall should allow traffic to the port and not block it, since the port 80 is used for HTTP and 443 for HTTPS is used by networks to connect.

## The Functional View:

The functional view involves in designing or getting to the functional elements, connectors, interfaces and the external entities required to develop a software, (4).

*Functional Elements (app.py)*:

This is the main software module which contains all the functions for creating the functionalities such as the static file handler, environment creator, the URL adapter and the server. It also contains extensive documentation on the usage of all the parameters. It includes several response-request cycles, handling exceptions etc.

*Connectors (__init__.py)/(cli.py):*

The *init* software code module contains imports all the Flask related functionality which is required for the application. It sets up the environment for building the web-based applications, by providing access to the Flask framework's features and utilities. It defines the various imports for flask related to classes, functions and modules. The *cli.py* module provides functionality to the user through the command line interface with the use of helper functions, the flask group subclass, the locate_app function and the decorators etc.

*Interface (API.RST/__init__.py):*

This software code module contains an application object which takes in certain parameters in order to implement the app. It contains central object which takes in parameters as other objects and modules. And acts as a central registry for the viewing functions, the URL and the template configuration. It is the framework which is required between the code and the end user. *API.rst* contains extensive documentation regarding this.

*External Entities:*

This involves the use of a webhosting service as previously explained during the stakeholder discussion. For the end user to see the working application, a host is required. For local applications, the programmer can create a local host and work through on their laptop/pc.

For MNC's, hosts (servers) which are available worldwide, therefore require cloud services like Microsoft Azure, Amazon Web Services etc.

Information View:

The information view focuses on how architecture stores, manipulates, manages, and distributes information, including aspects like content, structure, ownership, and latency, (4).

*Key Modules:*

The main modules, most relevant to the information view are:

*Sessions* can store information like user login states, preferences, and other data. The method for storing, retrieving, and expiring sessions directly relates to information management and security.

*globals.py:* This file defines proxies such as current_app, that are used to access information about current state of the application like who's currently using the app or what information they've asked for.

*config.py:* This file handles Flask's configuration management, which is crucial for understanding how and where various types of information (such as database connection strings, application secrets, etc.) are stored and managed.

*templating.py:* This file deals with the Jinja2 integration in Flask for templates. It's relevant to the information viewpoint because it shows how dynamic content is generated and served to users, including how data is passed to templates and transformed into HTML.

*wrappers.py:* This file wraps the objects of Werkzeug and defines the Request and Response classes in Flask. Understanding how information flows into and out of a Flask application requires an understanding of how requests and responses are handled. This includes how data is received from users, how it's processed, and how responses are constructed and sent back.

These files are essential in understanding how a Flask application manages, stores, and manipulates information within its environment. These components work together to handle user sessions, manage application and request-specific data, configure application settings, and process HTTP requests and responses.

Concurrency View

Flask uses threaded, event-driven and asynchronous concurrency models.

*Threaded* – A multithreading approach can be taken to allow for concurrent processing of multiple requests.

*Asynchronous* – Allows flask to continue executing other tasks while IO operations finish and while waiting for results to become available. Tasks can be scheduled to run when certain events occur e.g. timer expiring etc.

As with all concurrencies, it is important that proper execution of all actions occurs. With threading it is important that, if editing flask threads, shared variables are safely accessed to avoid race conditions and deadlocks (13). Similarly with asynchronous behaviour, it is important that blocking calls are avoided, as these will reduce the server's ability to handle concurrent requests dramatically.

Examples of files containing concurrency elements:

*Threads* – found in Web Server Gateway Interface (14)

Called in several code files

- app.py
- cli.py
- etc.

As well as this, asynchronous programming examples can also be found:

- views.py
- app.py
- ctx.py
- helpers.py
- templating.py
- etc.

Celery, (15), is used to handle the asynchronous task queuing. This allows for tasks to be completed while waiting for I/O.

Operational View

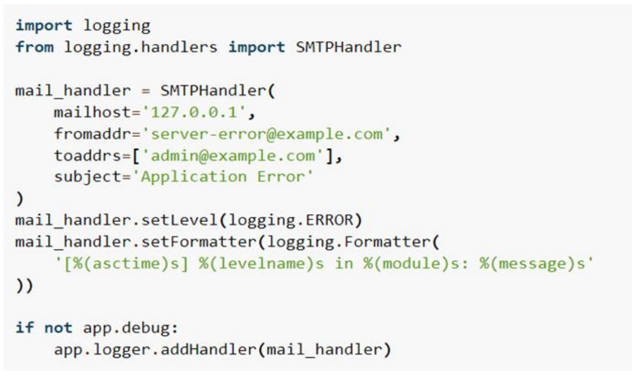
Flask has a support and maintenance mechanism where we should and can regularly be updated with all software's and versions of flask and python in our system, (8).

Since flask is open source, we can also leverage and take help from various of flask's online documentations, stack overflow and forums etc.

Flask is also administered by various features and software's which make sure there is no data leakage and tracks its logs, Tools such as Grafana and Prometheus helped detect slow response times and other performance issues.

For database administrators, Flask tools, like Flask-Migrate, handle database shifts and transfer data smoothly across different versions of the application. This often helps with the migration of the database.

If you still are facing issues and find bugs or errors in the flask application. You can configure this, Figure 5.



```python
import logging
from logging.handlers import SMTPHandler

mail_handler = SMTPHandler(
    mailhost='127.0.0.1',
    fromaddr='server-error@example.com',
    toaddrs=['admin@example.com'],
    subject='Application Error'
)
mail_handler.setLevel(logging.ERROR)
mail_handler.setFormatter(logging.Formatter(
    '[%(asctime)s] %(levelname)s in %(module)s: %(message)s'
))

if not app.debug:
    app.logger.addHandler(mail_handler)
```

Figure 5: SMTP Handler Code

An SMTP Handler in a Flask application is used via Python code to send emails for error-level logs. This helps set up with server and recipient information, it formats emails with log information and notifies administrators of errors at the same time.

You can use git to update your code for the flask application, you can also use MySQL to update databases.

**<u>Software Metrics</u>**

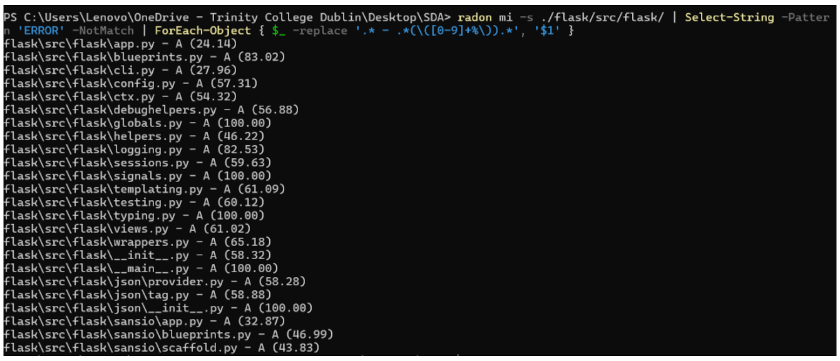
<u>Maintainability Index:</u>

Tool called radon was used, (16).

$$\mathit{Maintainability\ Index} = \max\left[0{,}100\,\frac{171 - 5.2\ln V - 0.23G - 16.2\ln L + 50\sin\left(\sqrt{2.4C}\right)}{171}\right]$$

Radon uses many metrics such as the Halstead Metrics (including volume), G is the cyclomatic complexity, L is the SLOC (source lines of code) and C would be percentage of commented lines.

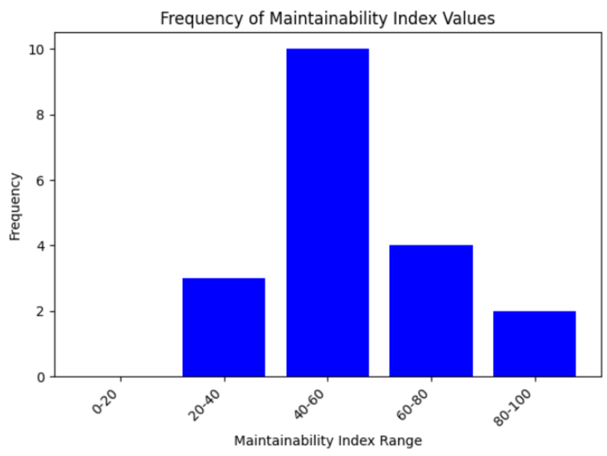
A rating of 0-10 would indicate a green rating (easy maintainability), 10-20 (moderate maintainability) and a rating higher than 20 indicates that maintaining that file would be easy, (17).



```
PS C:\Users\Lenovo\OneDrive - Trinity College Dublin\Desktop\SDA> radon mi -s ./flask/src/flask/ | Select-String -Patter
n 'ERROR' -NotMatch | ForEach-Object { $_ -replace '.* - .*(\([0-9]+%\)).*', '$1' }
flask\src\flask\app.py - A (24.14)
flask\src\flask\blueprints.py - A (83.02)
flask\src\flask\cli.py - A (27.96)
flask\src\flask\config.py - A (57.31)
flask\src\flask\ctx.py - A (54.32)
flask\src\flask\debughelpers.py - A (56.88)
flask\src\flask\globals.py - A (100.00)
flask\src\flask\helpers.py - A (46.22)
flask\src\flask\logging.py - A (82.53)
flask\src\flask\sessions.py - A (59.63)
flask\src\flask\signals.py - A (100.00)
flask\src\flask\templating.py - A (61.09)
flask\src\flask\testing.py - A (60.12)
flask\src\flask\typing.py - A (100.00)
flask\src\flask\views.py - A (61.02)
flask\src\flask\wrappers.py - A (65.18)
flask\src\flask\__init__.py - A (58.32)
flask\src\flask\__main__.py - A (100.00)
flask\src\flask\json\provider.py - A (58.28)
flask\src\flask\json\tag.py - A (58.88)
flask\src\flask\json\__init__.py - A (100.00)
flask\src\flask\sansio\app.py - A (32.87)
flask\src\flask\sansio\blueprints.py - A (46.99)
flask\src\flask\sansio\scaffold.py - A (43.83)
```

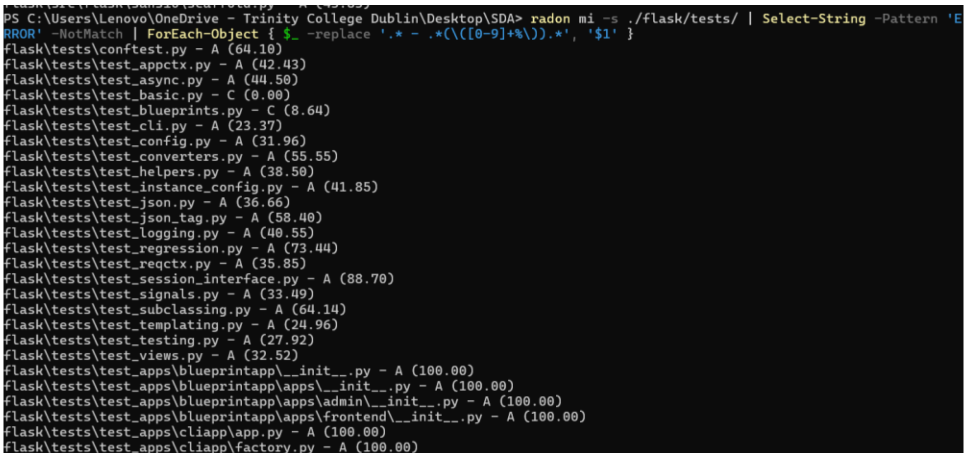
This is the maintainability index test performed on the directly of flask which contains all the required modules to run a flask application. It can be clearly seen from the ranks and their indexes that most of the files are easy to assess, modify and have high readability.

Figure 6: Maintainability Index Test of Flask



A bar graph, for the understanding of the same, has been produced.

Figure 8: Bar Graph Highlighting Frequency of Maintainability IndexValues



```
PS C:\Users\Lenovo\OneDrive - Trinity College Dublin\Desktop\SDA> radon mi -s ./flask/tests/ | Select-String -Pattern 'E
RROR' -NotMatch | ForEach-Object { $_ -replace '.* - .*(\([0-9]+%\)).*', '$1' }
flask\tests\conftest.py - A (64.10)
flask\tests\test_appctx.py - A (42.43)
flask\tests\test_async.py - A (44.50)
flask\tests\test_basic.py - C (0.00)
flask\tests\test_blueprints.py - C (8.64)
flask\tests\test_cli.py - A (23.37)
flask\tests\test_config.py - A (31.96)
flask\tests\test_converters.py - A (55.55)
flask\tests\test_helpers.py - A (38.50)
flask\tests\test_instance_config.py - A (41.85)
flask\tests\test_json.py - A (36.66)
flask\tests\test_json_tag.py - A (58.40)
flask\tests\test_logging.py - A (40.55)
flask\tests\test_regression.py - A (73.44)
flask\tests\test_reqctx.py - A (35.85)
flask\tests\test_session_interface.py - A (88.70)
flask\tests\test_signals.py - A (33.49)
flask\tests\test_subclassing.py - A (64.14)
flask\tests\test_templating.py - A (24.96)
flask\tests\test_testing.py - A (27.92)
flask\tests\test_views.py - A (32.52)
flask\tests\test_apps\blueprintapp\__init__.py - A (100.00)
flask\tests\test_apps\blueprintapp\apps\__init__.py - A (100.00)
flask\tests\test_apps\blueprintapp\apps\admin\__init__.py - A (100.00)
flask\tests\test_apps\blueprintapp\apps\frontend\__init__.py - A (100.00)
flask\tests\test_apps\cliapp\app.py - A (100.00)
flask\tests\test_apps\cliapp\factory.py - A (100.00)
```

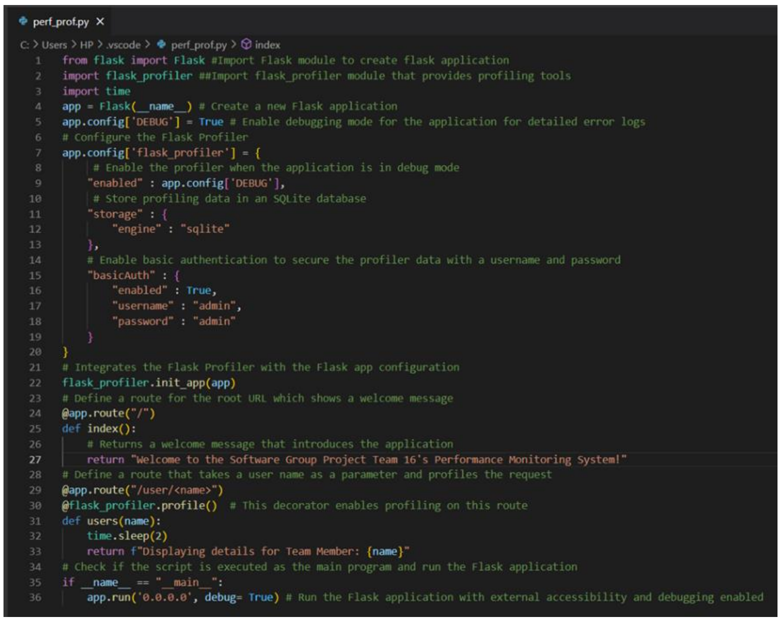
This is a similar test performed on the tests directory.

Figure 7: Similar Maintainability Index Test of Flask Tests

<u>Performance Profiling:</u>

- Using flask profiler, (18)
- Using memory profiler, (19)

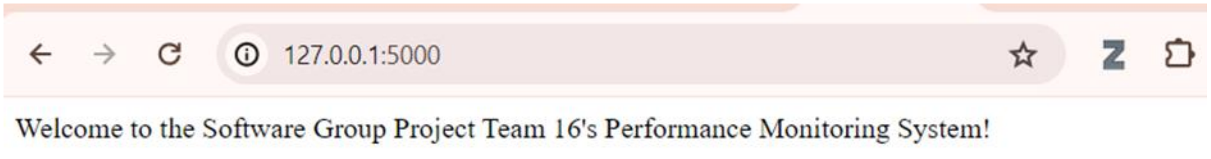
*Flask Profiler:*
The Flask Profiler is integrated to monitor the performance metrics of our flask application. We store profiling data in an SQLite database and protect it using basic authentication.



```python
from flask import Flask #Import Flask module to create flask application
import flask_profiler ##Import flask_profiler module that provides profiling tools
import time
app = Flask(__name__) # Create a new Flask application
app.config['DEBUG'] = True # Enable debugging mode for the application for detailed error logs
# Configure the Flask Profiler
app.config['flask_profiler'] = {
    # Enable the profiler when the application is in debug mode
    "enabled" : app.config['DEBUG'],
    # Store profiling data in an SQLite database
    "storage" : {
        "engine" : "sqlite"
    },
    # Enable basic authentication to secure the profiler data with a username and password
    "basicAuth" : {
        "enabled" : True,
        "username" : "admin",
        "password" : "admin"
    }
}
# Integrates the Flask Profiler with the Flask app configuration
flask_profiler.init_app(app)
# Define a route for the root URL which shows a welcome message
@app.route("/")
def index():
    # Returns a welcome message that introduces the application
    return "Welcome to the Software Group Project Team 16's Performance Monitoring System!"
# Define a route that takes a user name as a parameter and profiles the request
@app.route("/user/<name>")
@flask_profiler.profile()  # This decorator enables profiling on this route
def users(name):
    time.sleep(2)
    return f"Displaying details for Team Member: {name}"
# Check if the script is executed as the main program and run the Flask application
if __name__ == "__main__":
    app.run('0.0.0.0', debug= True) # Run the Flask application with external accessibility and debugging enabled
```

Figure 9: Flask Profiler Code

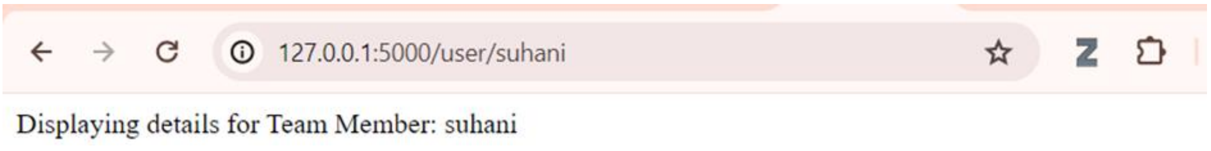
When the Flask server is launched, a welcome message appears when you go to the root URL [http://127.0.0.1:5000/](http://127.0.0.1:5000/)



Welcome to the Software Group Project Team 16's Performance Monitoring System!

Figure 10: Received Welcome Message when at Root URL

The user profiler endpoint /user/name dynamically generates a personalised message for the user when you go to the URL [127.0.0.1:5000/user/suhani](127.0.0.1:5000/user/suhani)

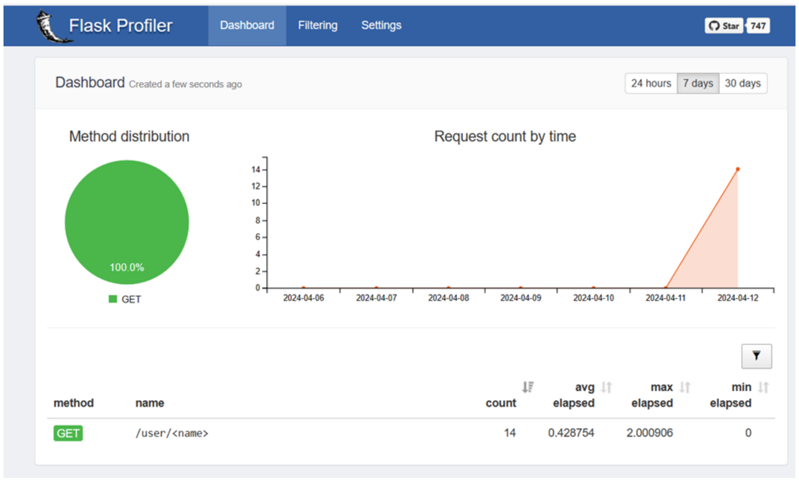


Displaying details for Team Member: suhani

Figure 11: Personalised User Message Generated

**Flask Profiling** when you go to the URL [127.0.0.1:5000/flask-profiler/](127.0.0.1:5000/flask-profiler/)

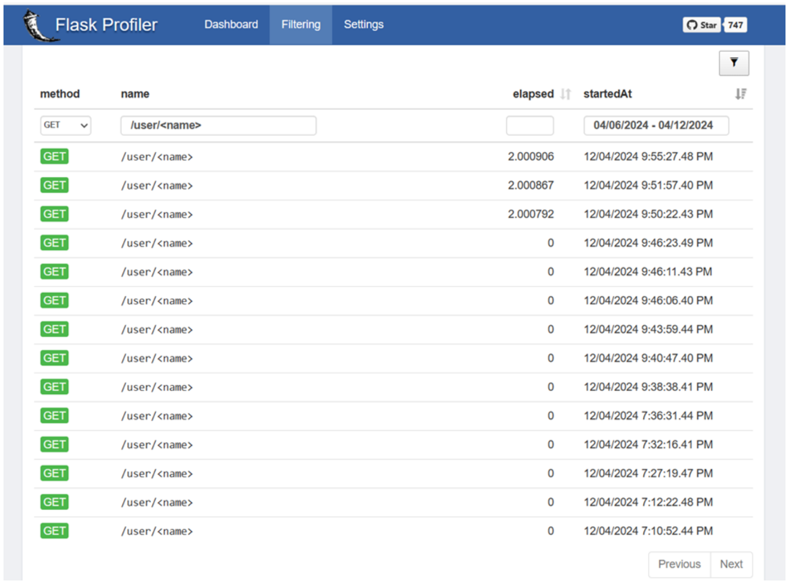
*Dashboard*
The dashboard validates our app's route configuration by displaying all traffic as GET requests. The request count graph's recent increases most likely represent testing. The average response time is 0.43 seconds with some requests taking up to 2 seconds which matches the delay in our endpoint /user/<name>.



| method | name | count | avg elapsed | max elapsed | min elapsed |
|---|---|---|---|---|---|
| GET | /user/<name> | 14 | 0.428754 | 2.000906 | 0 |

Figure 12: Flask Profiler Dashboard

*Filtering:*
The detailed log in the filtering view records several 2-second requests, reflecting our simulated delay. Requests with zero seconds indicate earlier tests without the delay, (18).

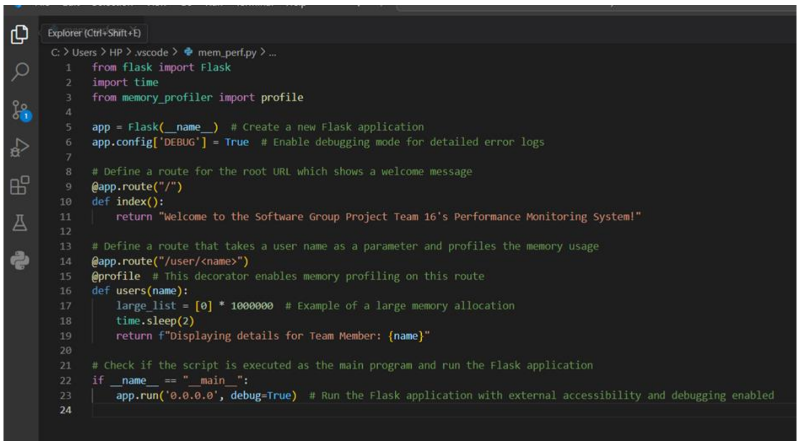


| method | name | elapsed | startedAt |
|---|---|---|---|
| GET | /user/<name> | | 04/06/2024 - 04/12/2024 |
| GET | /user/<name> | 2.000906 | 12/04/2024 9:55:27.48 PM |
| GET | /user/<name> | 2.000867 | 12/04/2024 9:51:57.40 PM |
| GET | /user/<name> | 2.000792 | 12/04/2024 9:50:22.43 PM |
| GET | /user/<name> | 0 | 12/04/2024 9:46:23.49 PM |
| GET | /user/<name> | 0 | 12/04/2024 9:46:11.43 PM |
| GET | /user/<name> | 0 | 12/04/2024 9:46:06.40 PM |
| GET | /user/<name> | 0 | 12/04/2024 9:43:59.44 PM |
| GET | /user/<name> | 0 | 12/04/2024 9:40:47.40 PM |
| GET | /user/<name> | 0 | 12/04/2024 9:38:38.41 PM |
| GET | /user/<name> | 0 | 12/04/2024 7:36:31.44 PM |
| GET | /user/<name> | 0 | 12/04/2024 7:32:16.41 PM |
| GET | /user/<name> | 0 | 12/04/2024 7:27:19.47 PM |
| GET | /user/<name> | 0 | 12/04/2024 7:12:22.48 PM |
| GET | /user/<name> | 0 | 12/04/2024 7:10:52.44 PM |

Figure 13: Flask Profiler Filtering

The Flask profiler tracks how long each request takes to process, helping identify slow or inefficient endpoints.

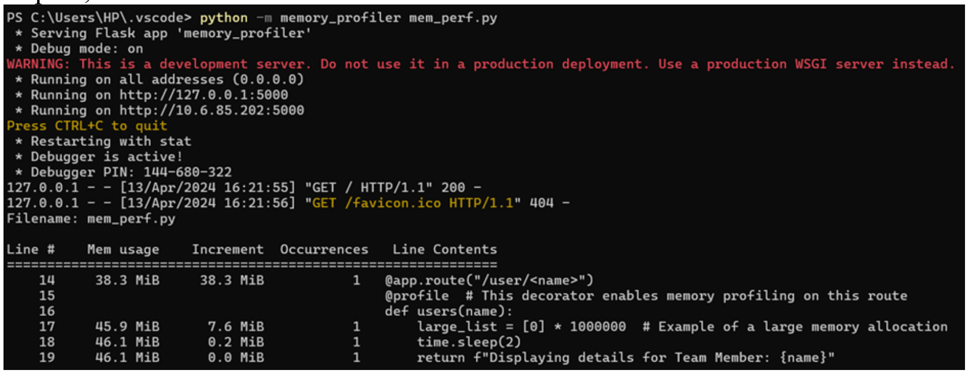
*Memory Profiler:*
We utilized the memory_profiler tool to review specific functions we aimed to profile within our Flask application. In order to do this, the functions were decorated using a @profile decorator.



```python
from flask import Flask
import time
from memory_profiler import profile

app = Flask(__name__)  # Create a new Flask application
app.config['DEBUG'] = True  # Enable debugging mode for detailed error logs

# Define a route for the root URL which shows a welcome message
@app.route("/")
def index():
    return "Welcome to the Software Group Project Team 16's Performance Monitoring System!"

# Define a route that takes a user name as a parameter and profiles the memory usage
@app.route("/user/<name>")
@profile  # This decorator enables memory profiling on this route
def users(name):
    large_list = [0] * 1000000  # Example of a large memory allocation
    time.sleep(2)
    return f"Displaying details for Team Member: {name}"

# Check if the script is executed as the main program and run the Flask application
if __name__ == "__main__":
    app.run('0.0.0.0', debug=True)  # Run the Flask application with external accessibility and debugging enabled
```

Figure 14: Memory Profiler Tool

Execution of the application with memory profiling was done using python -m memory_profiler mem_perf.py. The output shows that defining a large list with 1 million zeros in the Flask route /user/<name> increases memory usage by 7.6 MiB while the time.sleep(2) call has a minimal memory impact, with a 0.2 MiB increase.



```
PS C:\Users\HP\.vscode> python -m memory_profiler mem_perf.py
 * Serving Flask app 'memory_profiler'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.6.85.202:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 144-680-322
127.0.0.1 - - [13/Apr/2024 16:21:55] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Apr/2024 16:21:56] "GET /favicon.ico HTTP/1.1" 404 -
Filename: mem_perf.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    14     38.3 MiB     38.3 MiB           1   @app.route("/user/<name>")
    15                                         @profile  # This decorator enables memory profiling on this route
    16                                         def users(name):
    17     45.9 MiB      7.6 MiB           1       large_list = [0] * 1000000  # Example of a large memory allocation
    18     46.1 MiB      0.2 MiB           1       time.sleep(2)
    19     46.1 MiB      0.0 MiB           1       return f"Displaying details for Team Member: {name}"
```

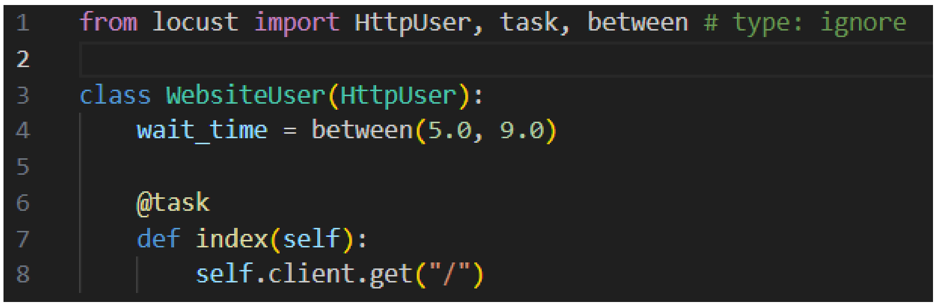
Figure 15: Memory Profiler Report

If we compare two popular Python web frameworks through memory profiling, several significant differences can be seen are. Flask, which is well-known for its ease of use and lightweight architecture, needs roughly 38 MiB of memory to run a basic web application as seen in the memory profiling table. Conversely, Django, which is considered as one of the best frameworks for reliable web development, requires approximately 60 MiB for similar tasks.

Load Testing:

Load testing is one of the most important tests which must be incorporated. To calculate the metric, the response time when the webserver is swarmed with users is measured, (9).

This can be accomplished by various software such as Apache JMeter for java applications, whereas for python in this case we used Locust, (20).



```python
from locust import HttpUser, task, between # type: ignore

class WebsiteUser(HttpUser):
    wait_time = between(5.0, 9.0)

    @task
    def index(self):
        self.client.get("/")
```

Figure 16: Locust Test Code

This is done by simply creating a website user every 5 and 9 seconds. This then sends a get request to the server which we are being connected to. Locust can create simultaneous users which is specified in their interface.

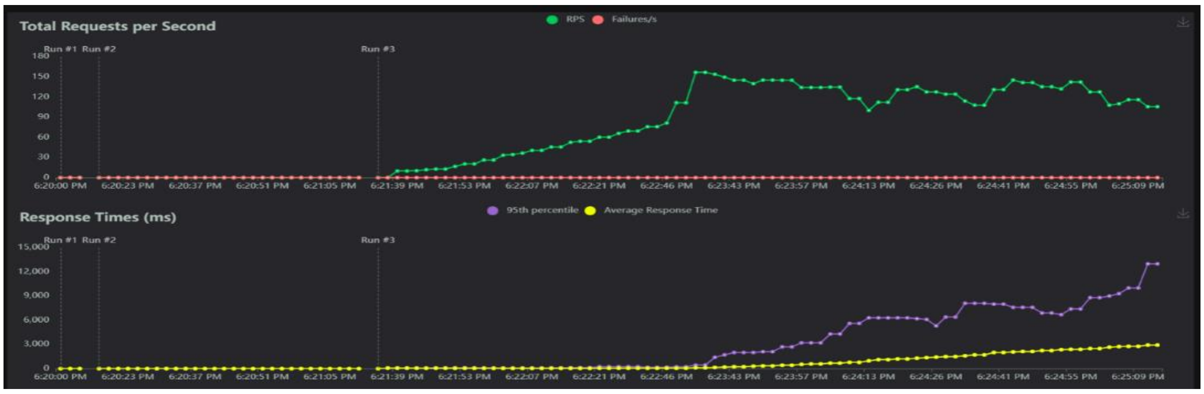


Figure 17: Graphs Produced by Locust

The graphs are plotted simultaneously. The 3rd curve (purple) is the one which needs to be checked, Figure 17. As can be clearly seen, the response times increase as the numbers of users increases. The 95th percentile is a more accurate statistic for testing as it is the industry norm with which the service providers are bounded by quality wise. These charts are like what we observed in our lectures.

The response times for the single server varies between around 40-50 ms per request. This is similar to the other famous web development Django which has a response time of around 45 ms.

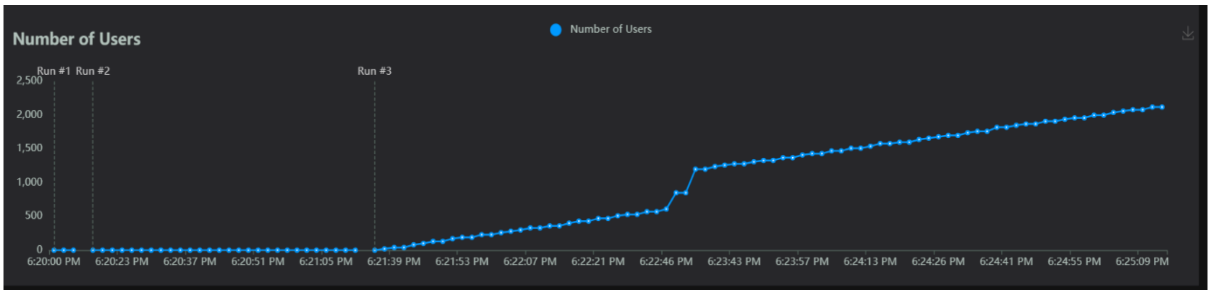


Figure 18: Response Time VS Number of Users

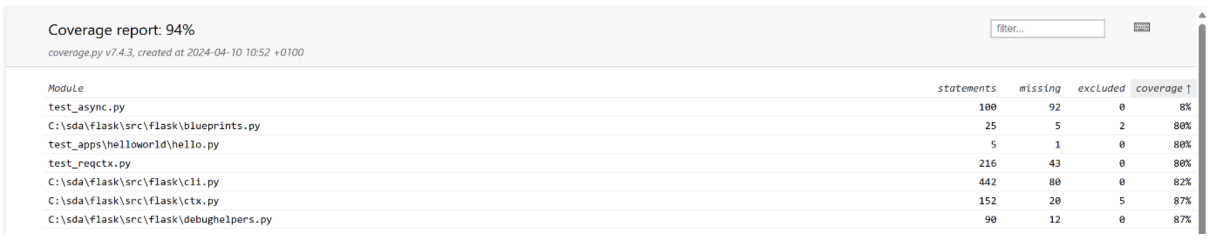
Unit Tests:
Analysing unit tests is crucial for verifying the expected behaviours of the Code Under Test (CUT). Testing also allows us to ensure that existing functionality is not broken during maintenance of code, they give the developers confidence that the code is functional and can be moved on from (9).

After analysing the repository, the following unit tests were decided on, for the following reasons:

1. Code Coverage
   This metric measures the how much of the codebase is covered by unit tests. Higher coverage indicates that the code has been tested effectively. A lower score here can indicate that an area may be under-tested and could contain bugs, (21).
2. Test Failure Rate
   This value represents the percentage of failed unit tests per commit/build. This value is used to represent the stability of code. If the value is regularly high, it could highlight that there should be a lack of confidence in the stability of the repository.
3. Test Execution Time
   This value is the average time to execute any unit test. This metric was chosen as having longer test execution times would slow the development of the code – which would have a large ripple effect on productivity, (22).
4. Test Duplication
   This metric is slightly related to metric 3. Test Duplication is the percentage of duplicated test code. Testing the same functionality repeatedly adds an excessive amount of overhead that will increase execution times, (23).

To gather data about these metrics from the flask repository, the pytest testing framework will be used, (24).

1. Code Coverage
   As seen in the figure, flask has a very good code coverage, with an average of 94%. A large number of the test codes have 100% code coverage (35/68) This shows that flasks testing is very comprehensive and suggests that the project is very unlikely to have bugs. However, it was seen that *test_async.py* had a very low coverage rate of 8%, which was unusual.



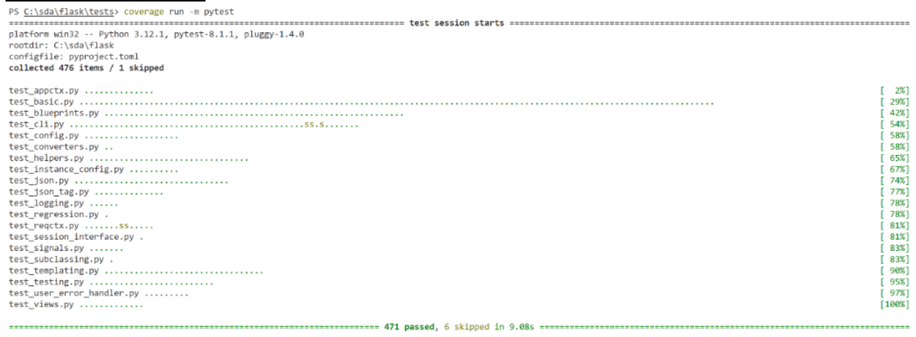
Coverage report: 94%

*coverage.py v7.4.3, created at 2024-04-10 10:52 +0100*

| Module | statements | missing | excluded | coverage ↑ |
|---|---|---|---|---|
| test_async.py | 100 | 92 | 0 | 8% |
| C:\sda\flask\src\flask\blueprints.py | 25 | 5 | 2 | 80% |
| test_apps\helloworld\hello.py | 5 | 1 | 0 | 80% |
| test_reqctx.py | 216 | 43 | 0 | 80% |
| C:\sda\flask\src\flask\cli.py | 442 | 80 | 0 | 82% |
| C:\sda\flask\src\flask\ctx.py | 152 | 20 | 5 | 87% |
| C:\sda\flask\src\flask\debughelpers.py | 90 | 12 | 0 | 87% |

Figure 19: Coverage Report Generated by pytest coverage

## 2. Test Failure Rate



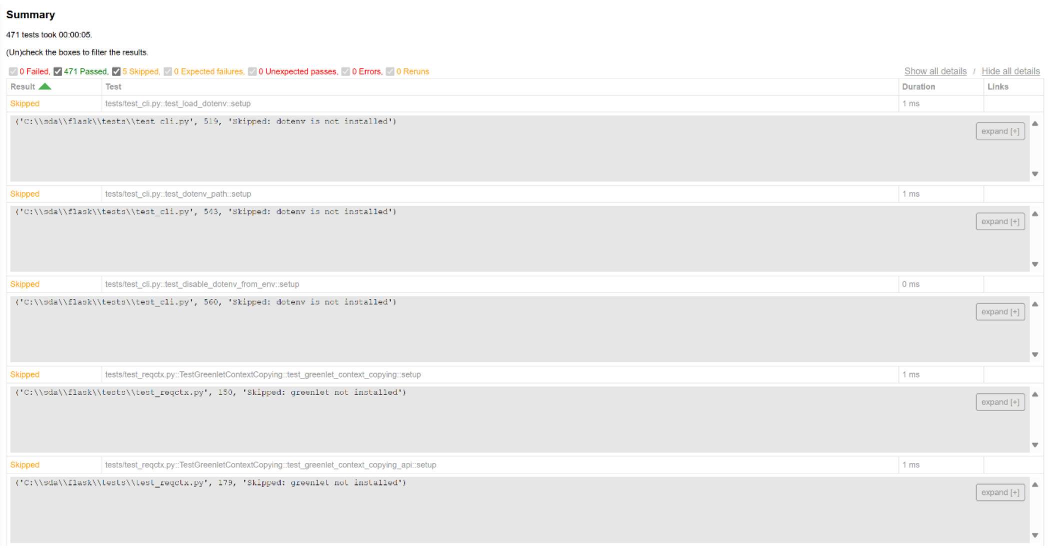
```
PS C:\sda\flask\tests> coverage run -m pytest
============================== test session starts ==============================
platform win32 -- Python 3.12.1, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\sda\flask
configfile: pyproject.toml
collected 476 items / 1 skipped

test_appctx.py ..............                                                 [  2%]
test_basic.py ..........................................................................................................................  [ 29%]
test_blueprints.py ............................................................  [ 42%]
test_cli.py ..............................................ss.s.......          [ 54%]
test_config.py ..................                                             [ 58%]
test_converters.py ..                                                         [ 58%]
test_helpers.py ................................                              [ 65%]
test_instance_config.py .........                                             [ 67%]
test_json.py ...............................                                  [ 74%]
test_json_tag.py ..............                                               [ 77%]
test_logging.py ......                                                        [ 78%]
test_regression.py .                                                          [ 78%]
test_reqctx.py .......ss.....                                                 [ 81%]
test_session_interface.py .                                                   [ 81%]
test_signals.py .......                                                       [ 83%]
test_subclassing.py .                                                         [ 83%]
test_templating.py ...............................                            [ 90%]
test_testing.py .........................                                     [ 95%]
test_user_error_handler.py .........                                          [ 97%]
test_views.py .............                                                   [100%]

======================== 471 passed, 6 skipped in 9.08s ========================
```

Figure 20: Coverage Report Progress Report

The pytest framework can be seen above. This reports that the repository at the time of testing had no failed tests:

$$Test\ Failure\ Rate = \frac{Number\ of\ Failed\ Tests}{Total\ Number\ of\ Tests} = \frac{0}{471+6}x100 = 0\%$$

It was also reported that 6 tests were skipped. These can be seen in the Figure 21 below:



**Summary**

471 tests took 00:00:05.

(Un)check the boxes to filter the results.

☐ 0 Failed, ☑ 471 Passed, ☑ 5 Skipped, ☐ 0 Expected failures, ☐ 0 Unexpected passes, ☐ 0 Errors, ☐ 0 Reruns

Show all details / Hide all details

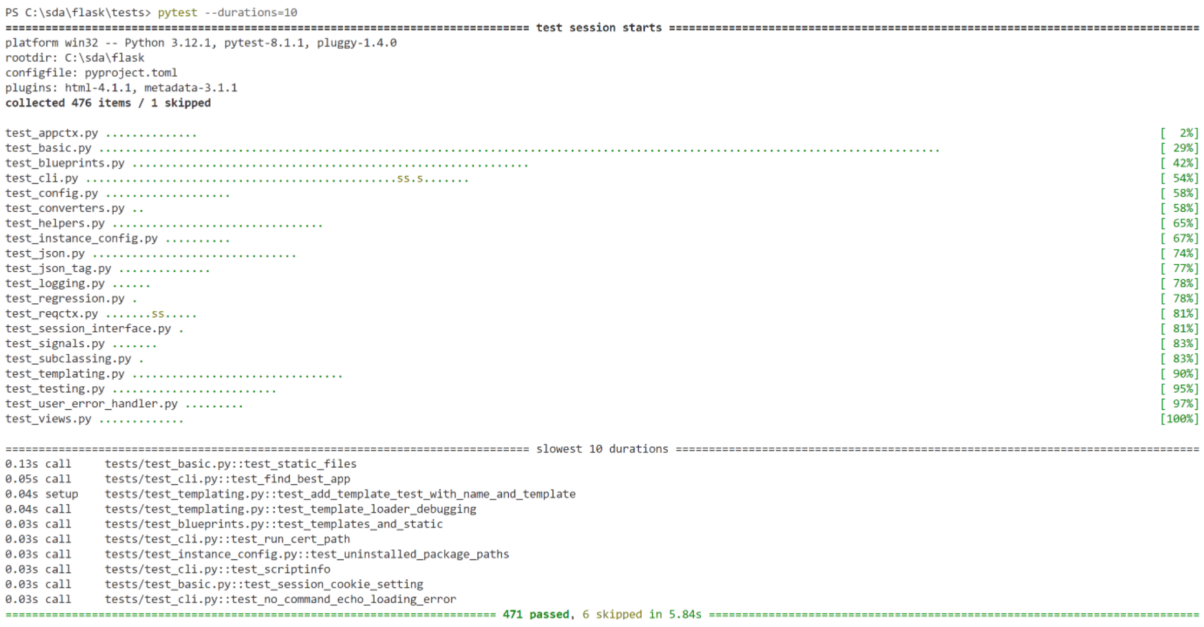
| Result ▲ | Test | Duration | Links |
|---|---|---|---|
| Skipped | tests/test_cli.py::test_load_dotenv::setup | 1 ms | |
| ('C:\\sda\\flask\\tests\\test_cli.py', 519, 'Skipped: dotenv is not installed') | | | |
| Skipped | tests/test_cli.py::test_dotenv_path::setup | 1 ms | |
| ('C:\\sda\\flask\\tests\\test_cli.py', 543, 'Skipped: dotenv is not installed') | | | |
| Skipped | tests/test_cli.py::test_disable_dotenv_from_env::setup | 0 ms | |
| ('C:\\sda\\flask\\tests\\test_cli.py', 560, 'Skipped: dotenv is not installed') | | | |
| Skipped | tests/test_reqctx.py::TestGreenletContextCopying::test_greenlet_context_copying::setup | 1 ms | |
| ('C:\\sda\\flask\\tests\\test_reqctx.py', 150, 'Skipped: greenlet not installed') | | | |
| Skipped | tests/test_reqctx.py::TestGreenletContextCopying::test_greenlet_context_copying_api::setup | 1 ms | |
| ('C:\\sda\\flask\\tests\\test_reqctx.py', 179, 'Skipped: greenlet not installed') | | | |

Figure 21: Skipped Tests

It is possible that these tests were skipped to ensure efficient testing, however this will be investigated.

## 3. Test Execution Time

From the previous pytest report, it was found that all tests were ran in 9.08 seconds.



```
PS C:\sda\flask\tests> pytest --durations=10
=========================================================================== test session starts ===========================================================================
platform win32 -- Python 3.12.1, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\sda\flask
configfile: pyproject.toml
plugins: html-4.1.1, metadata-3.1.1
collected 476 items / 1 skipped

test_appctx.py ..............                                                                                                                                   [  2%]
test_basic.py ..........................................................................................................................                       [ 29%]
test_blueprints.py ...........................................................                                                                                  [ 42%]
test_cli.py ...............................................ss.s.......                                                                                           [ 54%]
test_config.py ...................                                                                                                                              [ 58%]
test_converters.py ..                                                                                                                                           [ 58%]
test_helpers.py ................................                                                                                                                [ 65%]
test_instance_config.py ..........                                                                                                                              [ 67%]
test_json.py ...............................                                                                                                                    [ 74%]
test_json_tag.py ..............                                                                                                                                 [ 77%]
test_logging.py ......                                                                                                                                          [ 78%]
test_regression.py .                                                                                                                                            [ 78%]
test_reqctx.py .......ss.....                                                                                                                                   [ 81%]
test_session_interface.py .                                                                                                                                     [ 81%]
test_signals.py .......                                                                                                                                         [ 83%]
test_subclassing.py .                                                                                                                                           [ 83%]
test_templating.py ................................                                                                                                             [ 90%]
test_testing.py .........................                                                                                                                       [ 95%]
test_user_error_handler.py .........                                                                                                                            [ 97%]
test_views.py .............                                                                                                                                     [100%]

========================================================================== slowest 10 durations ===========================================================================
0.13s call     tests/test_basic.py::test_static_files
0.05s call     tests/test_cli.py::test_find_best_app
0.04s setup    tests/test_templating.py::test_add_template_test_with_name_and_template
0.04s call     tests/test_templating.py::test_template_loader_debugging
0.03s call     tests/test_blueprints.py::test_templates_and_static
0.03s call     tests/test_cli.py::test_run_cert_path
0.03s call     tests/test_instance_config.py::test_uninstalled_package_paths
0.03s call     tests/test_cli.py::test_scriptinfo
0.03s call     tests/test_basic.py::test_session_cookie_setting
0.03s call     tests/test_cli.py::test_no_command_echo_loading_error
===================================================================== 471 passed, 6 skipped in 5.84s ======================================================================
```

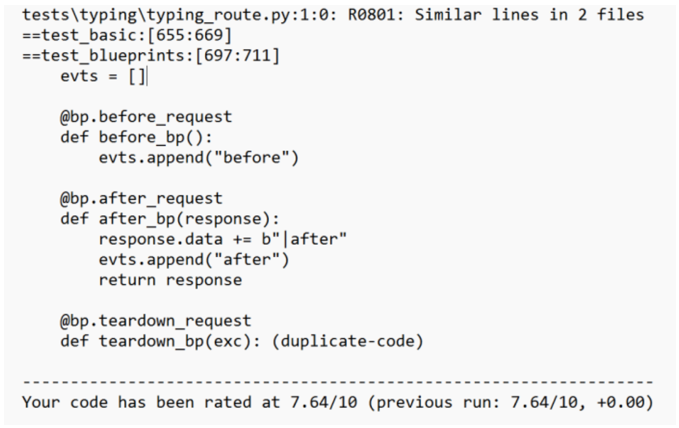
Figure 22: pytest Report Showing 10 Slowest Execution Times

After running the frameworks built-in durations option, the 10 slowest execution times were found, Figure 22. It can be seen that the total execution time varies between runs.

The 9.08 second test execution time could be considered slow for Flask, however since this value was halved in the next run, it is not considered to be an issue.

4. <u>Test Duplication</u>

By running static analysis of the test codes using pylint, the following similarities were found.



```
tests\typing\typing_route.py:1:0: R0801: Similar lines in 2 files
==test_basic:[655:669]
==test_blueprints:[697:711]
    evts = []

    @bp.before_request
    def before_bp():
        evts.append("before")

    @bp.after_request
    def after_bp(response):
        response.data += b"|after"
        evts.append("after")
        return response

    @bp.teardown_request
    def teardown_bp(exc): (duplicate-code)

-----------------------------------------------------------------
Your code has been rated at 7.64/10 (previous run: 7.64/10, +0.00)
```

Figure 23: Analysis of Duplicates from pylint Report

Since only 2 similar lines were found, it is unlikely that these are causing any excessive amounts of overhead while running the code. As well as this, the report results in a rating of 7.64/10. This is a good result however it shows that there is room for improvement. Many of the "issues" causing this rating were trivial, such as unused arguments etc.

## <u>**Contributions and Documentations:**</u>

### <u>Literature Review</u>

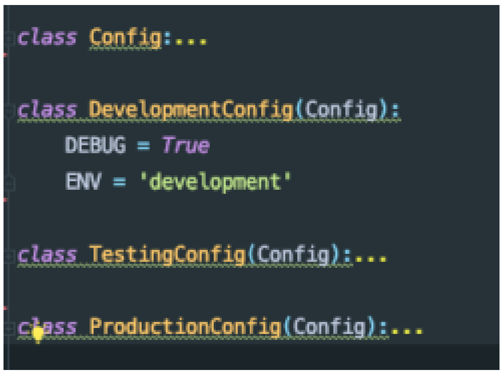
We are going to explain and understand the relevant aspects of the research paper written by Devndra Ghimire, on a comparative study of flask and Django (25).

Flask allows developers to organize their web applications flexibly. While it doesn't force any specific structure. Flask utilizes Jinja2 as its template engine. Jinja2 allows Python-like expressions for rapid development of dynamic web pages, providing powerful features like template inheritance and sandboxed execution, its main idea is to have more functionality with less minimal setup of its own. It functions on WSGI Application Framework.

<u>Security:</u> One of the softwares that help with the security issues is ORM (Object Relational Model). This helps with SQL injections, they have data validation models which builds on the well-formed data intercepts with the SQL. ORM libraries also have multiple mechanisms which make sure issues regarding data do not arise. Another software, CSRF, ensures protection can be easily implemented with extensions like Flask-WTF, which includes a CSRF module and tokens. Flask checks the tokens from the form and then automatically validates it.

<u>Configurations:</u> There are specific configurations for each task that is performed in the flask application like Deployment, Testing and Production configurations. Development configurations might enable debug mode, testing configurations could connect to a different database or mock services, and production settings would tighten security measures and optimize performance.

This is how different configurations are put into classes:



```
class Config:...

class DevelopmentConfig(Config):
    DEBUG = True
    ENV = 'development'

class TestingConfig(Config):...

class ProductionConfig(Config):...
```

Figure 24: Configurations in Flask

<u>Database:</u> Flask has a flexible database system, it wants to make sure that its light and simple and also allows us to use of our choice, as it helps integrates with third-party ORMs like SQL Alchemy for interacting with databases such as PostgreSQL or MySQL.

<u>Caching:</u> Initially the project was built without any caching mechanism since it was a minimum value product, eventually when it became of importance it provided with options such as Filesystem Cache, Redis Cache, UWSGI Cache, Memcached Cache, and SASLM Cache.

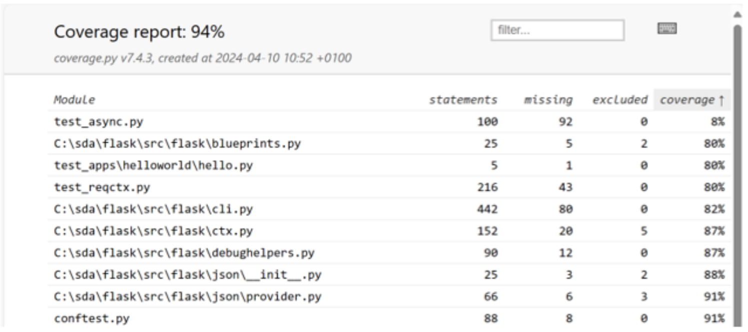
The filesystem cache stores data in the file system, it's good for single setup users but not scalable.

Unlike the filesystem one, the Redis Cache, is highly scalable and offers high performance systems with support of complex data types.

<u>Routing:</u> This involves mapping URLs to Python functions which handles web requests, A URL pattern is mapped by the @app route decorator to a corresponding function, which when called, performs the logic contained in that function. Views are protected by the @login_required decorator, which makes sure that only authorized users can activate them.

<u>Proposed optimizations for poorly tested or slow features:</u>

As discussed earlier, while carrying out unit testing, it was noticed that the test file "test_async.py" had a very low code coverage. It was seen that its coverage was the lowest by 72%. This stood out, as all other files in the flask repository had a code coverage of above 80%. Figure 25



Coverage report: 94%

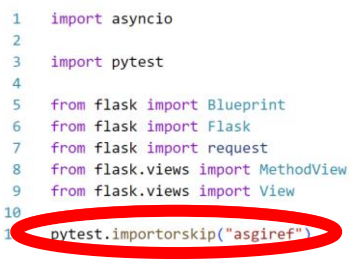
*coverage.py v7.4.3, created at 2024-04-10 10:52 +0100*

| Module | statements | missing | excluded | coverage ↑ |
|---|---|---|---|---|
| test_async.py | 100 | 92 | 0 | 8% |
| C:\sda\flask\src\flask\blueprints.py | 25 | 5 | 2 | 80% |
| test_apps\helloworld\hello.py | 5 | 1 | 0 | 80% |
| test_reqctx.py | 216 | 43 | 0 | 80% |
| C:\sda\flask\src\flask\cli.py | 442 | 80 | 0 | 82% |
| C:\sda\flask\src\flask\ctx.py | 152 | 20 | 5 | 87% |
| C:\sda\flask\src\flask\debughelpers.py | 90 | 12 | 0 | 87% |
| C:\sda\flask\src\flask\json__init__.py | 25 | 3 | 2 | 88% |
| C:\sda\flask\src\flask\json\provider.py | 66 | 6 | 3 | 91% |
| conftest.py | 88 | 8 | 0 | 91% |

Figure 25: Initial pytest coverage Report

To investigate this, file test_async.py was analysed in-depth to figure out why the coverage was low.

As seen in the Figure 26, if the “asgiref” module is not available, tests will be skipped. For this reason, the asgiref module was downloaded, (26).

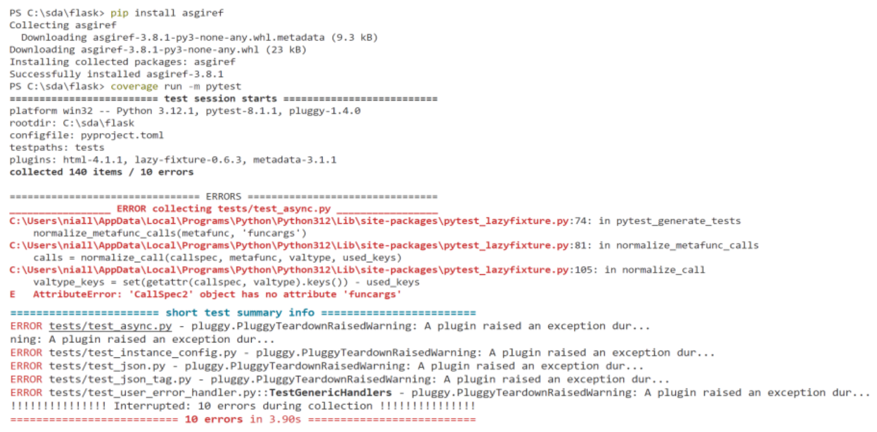


```
import asyncio

import pytest

from flask import Blueprint
from flask import Flask
from flask import request
from flask.views import MethodView
from flask.views import View

pytest.importorskip("asgiref")
```

Figure 26: Excerpt from tests_async.py

Also mentioned in the Unit Testing section, 6 other tests were skipped. From looking at the generated .html file, it was seen these were skipped due to missing libraries, (greenlet & dotenv). These were also installed, (27), (28).

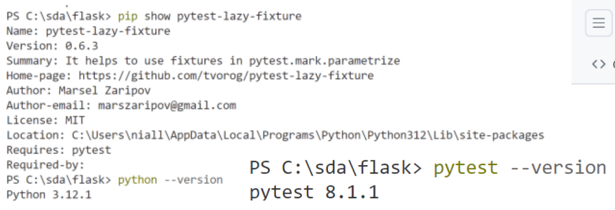
Following this, the coverage report was regenerated. In this run, 10 errors were found.



```
PS C:\sda\flask> pip install asgiref
Collecting asgiref
  Downloading asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)
Downloading asgiref-3.8.1-py3-none-any.whl (23 kB)
Installing collected packages: asgiref
Successfully installed asgiref-3.8.1
PS C:\sda\flask> coverage run -m pytest
========================= test session starts ==========================
platform win32 -- Python 3.12.1, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\sda\flask
configfile: pyproject.toml
testpaths: tests
plugins: html-4.1.1, lazy-fixture-0.6.3, metadata-3.1.1
collected 140 items / 10 errors

================================ ERRORS ================================
_________________ ERROR collecting tests/test_async.py _________________
C:\Users\niall\AppData\Local\Programs\Python\Python312\Lib\site-packages\pytest_lazyfixture.py:74: in pytest_generate_tests
    normalize_metafunc_calls(metafunc, 'funcargs')
C:\Users\niall\AppData\Local\Programs\Python\Python312\Lib\site-packages\pytest_lazyfixture.py:81: in normalize_metafunc_calls
    calls = normalize_call(callspec, metafunc, valtype, used_keys)
C:\Users\niall\AppData\Local\Programs\Python\Python312\Lib\site-packages\pytest_lazyfixture.py:105: in normalize_call
    valtype_keys = set(getattr(callspec, valtype).keys()) - used_keys
E   AttributeError: 'CallSpec2' object has no attribute 'funcargs'
======================== short test summary info =======================
ERROR tests/test_async.py - pluggy.PluggyTeardownRaisedWarning: A plugin raised an exception dur...
ning: A plugin raised an exception dur...
ERROR tests/test_instance_config.py - pluggy.PluggyTeardownRaisedWarning: A plugin raised an exception dur...
ERROR tests/test_json.py - pluggy.PluggyTeardownRaisedWarning: A plugin raised an exception dur...
ERROR tests/test_json_tag.py - pluggy.PluggyTeardownRaisedWarning: A plugin raised an exception dur...
ERROR tests/test_user_error_handler.py::TestGenericHandlers - pluggy.PluggyTeardownRaisedWarning: A plugin raised an exception dur...
!!!!!!!!!!!!!!! Interrupted: 10 errors during collection !!!!!!!!!!!!!!!
========================== 10 errors in 3.90s ==========================
```

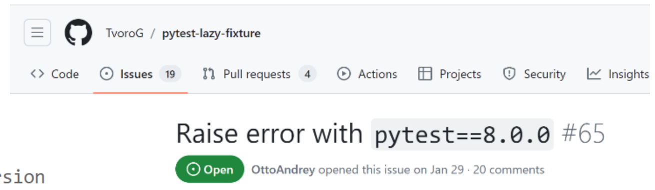
Figure 27: Coverage Report Containing Errors

After further investigation of both the *test_async.py* file and the error report, it was found that these errors were caused by a compatibility issue with pytest-lazy-fixture, which had been used earlier static analysis, (29).



```
PS C:\sda\flask> pip show pytest-lazy-fixture
Name: pytest-lazy-fixture
Version: 0.6.3
Summary: It helps to use fixtures in pytest.mark.parametrize
Home-page: https://github.com/tvorog/pytest-lazy-fixture
Author: Marsel Zaripov
Author-email: marszaripov@gmail.com
License: MIT
Location: C:\Users\niall\AppData\Local\Programs\Python\Python312\Lib\site-packages
Requires: pytest
Required-by:
PS C:\sda\flask> python --version
Python 3.12.1
```

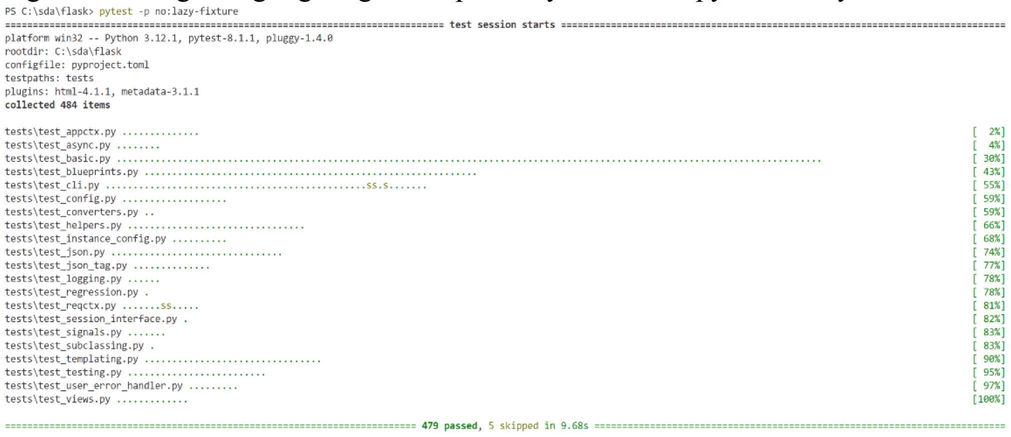
```
PS C:\sda\flask> pytest --version
pytest 8.1.1
```



Raise error with `pytest==8.0.0` #65

OttoAndrey opened this issue on Jan 29 · 20 comments

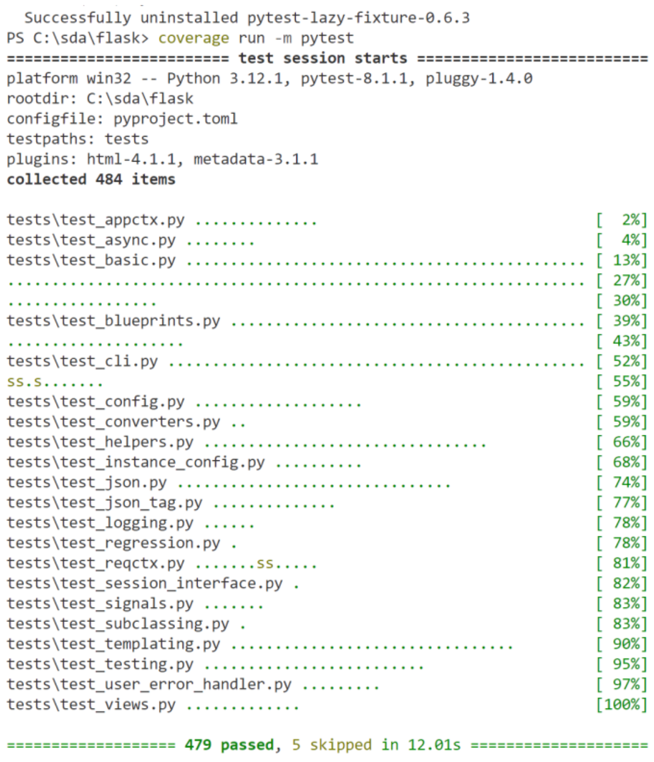
Figure 28: Images Highlighting Incompatibility Issue with pytest & lazy-fixture



```
PS C:\sda\flask> pytest -p no:lazy-fixture
=============================================================================== test session starts ===============================================================================
platform win32 -- Python 3.12.1, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\sda\flask
configfile: pyproject.toml
testpaths: tests
plugins: html-4.1.1, metadata-3.1.1
collected 484 items

tests\test_appctx.py ..............                                                                                                                                         [  2%]
tests\test_async.py ........                                                                                                                                                [  4%]
tests\test_basic.py ..............................................................................................................................                              [ 30%]
tests\test_blueprints.py ..............................................................                                                                                     [ 43%]
tests\test_cli.py ..............................................ss.s.......                                                                                                 [ 55%]
tests\test_config.py ...................                                                                                                                                    [ 59%]
tests\test_converters.py ..                                                                                                                                                 [ 59%]
tests\test_helpers.py ................................                                                                                                                      [ 66%]
tests\test_instance_config.py ..........                                                                                                                                    [ 68%]
tests\test_json.py ...............................                                                                                                                          [ 74%]
tests\test_json_tag.py ..............                                                                                                                                       [ 77%]
tests\test_logging.py ......                                                                                                                                                [ 78%]
tests\test_regression.py .                                                                                                                                                  [ 78%]
tests\test_reqctx.py .......ss.....                                                                                                                                         [ 81%]
tests\test_session_interface.py .                                                                                                                                           [ 82%]
tests\test_signals.py .......                                                                                                                                               [ 83%]
tests\test_subclassing.py .                                                                                                                                                 [ 83%]
tests\test_templating.py ................................                                                                                                                   [ 90%]
tests\test_testing.py .........................                                                                                                                             [ 95%]
tests\test_user_error_handler.py .........                                                                                                                                  [ 97%]
tests\test_views.py .............                                                                                                                                           [100%]

========================================================================= 479 passed, 5 skipped in 9.68s ==========================================================================
```

Figure 29: Showing tests running without lazy fixture, highlighting it was causing the issue

After trying previous versions of pylint, lazy-fixture was uninstalled, Figure 30.



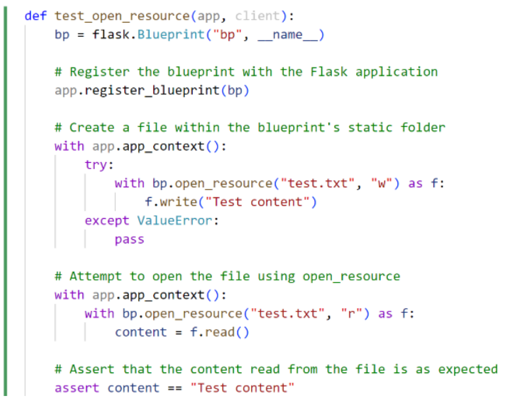
```
  Successfully uninstalled pytest-lazy-fixture-0.6.3
PS C:\sda\flask> coverage run -m pytest
======================== test session starts =========================
platform win32 -- Python 3.12.1, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\sda\flask
configfile: pyproject.toml
testpaths: tests
plugins: html-4.1.1, metadata-3.1.1
collected 484 items

tests\test_appctx.py .............                                [  2%]
tests\test_async.py .......                                       [  4%]
tests\test_basic.py ........................................      [ 13%]
...............................................................   [ 27%]
.................                                                 [ 30%]
tests\test_blueprints.py ........................................ [ 39%]
....................                                              [ 43%]
tests\test_cli.py ...............................................  [ 52%]
ss.s.......                                                       [ 55%]
tests\test_config.py ..................                           [ 59%]
tests\test_converters.py ..                                       [ 59%]
tests\test_helpers.py ...............................             [ 66%]
tests\test_instance_config.py ..........                          [ 68%]
tests\test_json.py ..............................                 [ 74%]
tests\test_json_tag.py ..............                             [ 77%]
tests\test_logging.py ......                                      [ 78%]
tests\test_regression.py .                                        [ 78%]
tests\test_reqctx.py .......ss.....                               [ 81%]
tests\test_session_interface.py .                                 [ 82%]
tests\test_signals.py .......                                     [ 83%]
tests\test_subclassing.py .                                       [ 83%]
tests\test_templating.py ...............................          [ 90%]
tests\test_testing.py ........................                    [ 95%]
tests\test_user_error_handler.py .........                        [ 97%]
tests\test_views.py ............                                  [100%]

================== 479 passed, 5 skipped in 12.01s ===================
```

Figure 30: Uninstalling lazy-fixture

This resulted in test_async.py's coverage being raised to 100%.

After this, the lowest test coverage value was 80%. By adding code to test other cases, we were able to successfully increase this test coverage to 92%. This also resulted in a 3% increase in the overall test coverage from 94% to 97%.

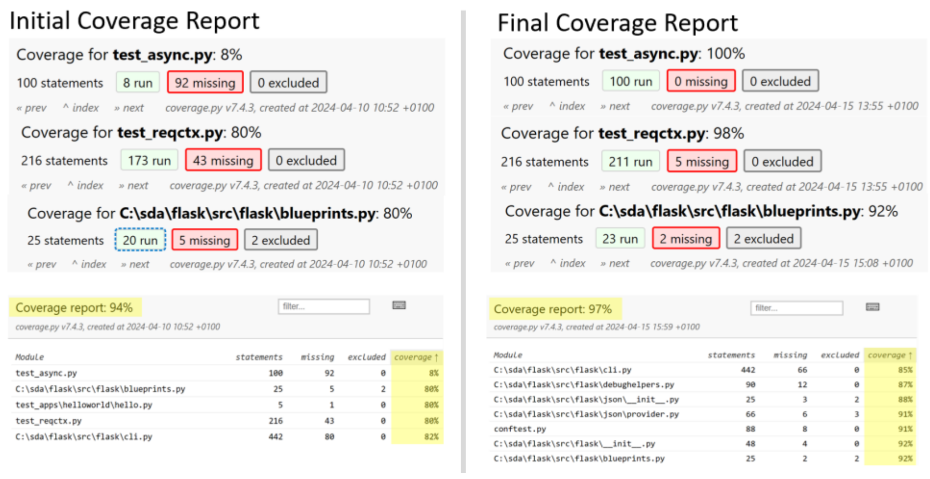
An example of the code added to existing test files is shown here.



```python
def test_open_resource(app, client):
    bp = flask.Blueprint("bp", __name__)

    # Register the blueprint with the Flask application
    app.register_blueprint(bp)

    # Create a file within the blueprint's static folder
    with app.app_context():
        try:
            with bp.open_resource("test.txt", "w") as f:
                f.write("Test content")
        except ValueError:
            pass

    # Attempt to open the file using open_resource
    with app.app_context():
        with bp.open_resource("test.txt", "r") as f:
            content = f.read()

    # Assert that the content read from the file is as expected
    assert content == "Test content"
```

Figure 31: Example of Additions to Test Codes

A quantitative comparison between the Initial & Final Coverage Reports can be seen here.



Initial Coverage Report

Coverage for **test_async.py**: 8%

100 statements | 8 run | 92 missing | 0 excluded

« prev ^ index » next coverage.py v7.4.3, created at 2024-04-10 10:52 +0100

Coverage for **test_reqctx.py**: 80%

216 statements | 173 run | 43 missing | 0 excluded

« prev ^ index » next coverage.py v7.4.3, created at 2024-04-10 10:52 +0100

Coverage for **C:\sda\flask\src\flask\blueprints.py**: 80%

25 statements | 20 run | 5 missing | 2 excluded

« prev ^ index » next coverage.py v7.4.3, created at 2024-04-10 10:52 +0100

Coverage report: 94%

coverage.py v7.4.3, created at 2024-04-10 10:52 +0100

| Module | statements | missing | excluded | coverage ↑ |
|---|---|---|---|---|
| test_async.py | 100 | 92 | 0 | 8% |
| C:\sda\flask\src\flask\blueprints.py | 25 | 5 | 2 | 80% |
| test_apps\helloworld\hello.py | 5 | 1 | 0 | 80% |
| test_reqctx.py | 216 | 43 | 0 | 80% |
| C:\sda\flask\src\flask\cli.py | 442 | 80 | 0 | 82% |

Final Coverage Report

Coverage for **test_async.py**: 100%

100 statements | 100 run | 0 missing | 0 excluded

« prev ^ index » next coverage.py v7.4.3, created at 2024-04-15 13:55 +0100

Coverage for **test_reqctx.py**: 98%

216 statements | 211 run | 5 missing | 0 excluded

« prev ^ index » next coverage.py v7.4.3, created at 2024-04-15 13:55 +0100

Coverage for **C:\sda\flask\src\flask\blueprints.py**: 92%

25 statements | 23 run | 2 missing | 2 excluded

« prev ^ index » next coverage.py v7.4.3, created at 2024-04-15 15:08 +0100

Coverage report: 97%

coverage.py v7.4.3, created at 2024-04-15 15:59 +0100

| Module | statements | missing | excluded | coverage ↑ |
|---|---|---|---|---|
| C:\sda\flask\src\flask\cli.py | 442 | 66 | 0 | 85% |
| C:\sda\flask\src\flask\debughelpers.py | 90 | 12 | 0 | 87% |
| C:\sda\flask\src\flask\json\__init__.py | 25 | 3 | 2 | 88% |
| C:\sda\flask\src\flask\json\provider.py | 66 | 6 | 3 | 91% |
| conftest.py | 88 | 8 | 0 | 91% |
| C:\sda\flask\src\flask\__init__.py | 48 | 4 | 0 | 92% |
| C:\sda\flask\src\flask\blueprints.py | 25 | 2 | 2 | 92% |

Figure 32: Comparison between Initial Coverage & Final Coverage

<u>Code Clones & Maintainability</u>
Upon analysing the Flask repository using Code Climate, (30), the analysis showed zero instances of duplication, indicating that as of the last analysis, there are no significant code clones inside this Flask application that could cause additional maintenance or security concerns. The absence of duplication is indicative of a healthier, cleaner codebase which can be easier to maintain and evolve.

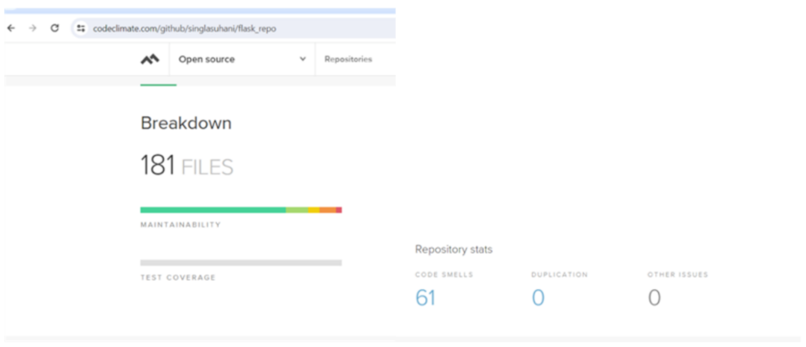


Figure 33: Analysis of Code Clones Using Code Climate

The code smells were caused by inconsistencies between Code Climates standards & the Flask Repositories Standards, such as over 250 lines of code in a file etc.

*Are all developers focusing on a single file?*
When developers start initially, they would focus on single filed application since the architecture and its complexity is relatively simple.

As more applications and complexity increases, which means increase in the models, controllers and configurations, there might be blueprints that are introduced to take in account of the complex structure of the file system. This is where developers start working on different files in the same application to enhance maintainability, readability and scalability of the code.

**<u>Conclusion:</u>**
Our group project involved an analysis of the Flask framework. It highlighted its strengths in flexibility and community support.

Flask's minimalist approach allows developers to select extensions based on the demands of the project. Our stakeholder analysis highlighted the active and supportive Flask community, crucial for ongoing maintenance and enhancement of the framework. The created views provided the group with a clear understanding of the relevant areas when undergoing analysis/ improvements.

Through our software metrics study, we were able to see strong maintainability and robust testing throughout the Flask ecosystem, demonstrating its applicability for both small and enterprise-level applications.

We believe our efforts at improving documentation will improve developer experience. Our contributions saw an increase in 3% overall test coverage (Previously 94% now 97% coverage) with all individual test coverages now being above 85%. This is a slight, yet significant improvement in the coverage, meaning that the code is more extensively tested and therefore, is likely more robust and reliable.

**Individual Contributions:**

**Week 1:** Project Setup and Initial Analysis

*Raaghav:* Identify stakeholders and outline the functional architectural view.

*Suhani:* Outline the context and information views.

*Niall:* Outline the development and concurrency views.

*Lakshaya:* Outline the deployment and operational views.

**Week 2:** Software Metrics Evaluation

*Raaghav:* Perform Maintainability Index and Load Testing.

*Suhani*: Perform Performance Profiling.

*Niall*: Perform Unit Testing.

**Week 3:** Research and Improvement Proposals

*Lakshaya:* Analyze relevant research papers and identify areas of single developer focus.

*Niall:* Propose optimizations for poorly tested or slow features.

*Suhani:* Identify any code clones.

*Raaghav:* Identify the maintainability index related issues.

**Week 4:** Report and Presentation Preparation

*Suhani & Lakshaya:* Draft and finalize the project report, focusing on introduction, analysis and conclusion.

*Raaghav & Niall:* Prepare the project presentation.

**References**

1. Keep Developing! — Flask Documentation (2.3.x) [Internet]. [cited 2024 Apr 17]. Available from: https://flask.palletsprojects.com/en/2.3.x/tutorial/next/
2. Pallets [Internet]. [cited 2024 Apr 17]. Werkzeug. Available from: https://palletsprojects.com/p/werkzeug/
3. Pallets [Internet]. [cited 2024 Apr 17]. Jinja. Available from: https://palletsprojects.com/p/jinja/
4. Rozanski N, Woods E. Software Systems Architecture. 2nd ed. Pearson Education; 2011.
5. BSD-3-Clause License — Flask Documentation (2.3.x) [Internet]. [cited 2024 Apr 17]. Available from: https://flask.palletsprojects.com/en/2.3.x/license/
6. Python.org [Internet]. [cited 2024 Apr 17]. Welcome to Python.org. Available from: https://www.python.org/psf-landing/
7. Pocoo [Internet]. [cited 2024 Apr 17]. Available from: https://www.pocoo.org/
8. pallets/flask: The Python micro framework for building web applications. [Internet]. [cited 2024 Apr 17]. Available from: https://github.com/pallets/flask
9. Ventresque A. Lecture 4: Code Quality (2) [Internet]. Available from: https://tcd.blackboard.com/
10. Jinja — Jinja Documentation (3.1.x) [Internet]. [cited 2024 Mar 25]. Available from: https://jinja.palletsprojects.com/en/3.1.x/

11. Dugar D. Medium. 2023 [cited 2024 Mar 25]. Jinja2 Explained in 5 Minutes! Available from: https://codeburst.io/jinja-2-explained-in-5-minutes-88548486834e

12. Blueprints and Views — Flask Documentation (2.3.x) [Internet]. [cited 2024 Mar 25]. Available from: https://flask.palletsprojects.com/en/2.3.x/tutorial/views/

13. Nguyen Q. Mastering Concurrency in Python. Packt Publishing; 2018.

14. Werkzeug — Werkzeug Documentation (3.0.x) [Internet]. [cited 2024 Apr 17]. Available from: https://werkzeug.palletsprojects.com/en/3.0.x/

15. Introduction to Celery — Celery 5.3.6 documentation [Internet]. [cited 2024 Apr 17]. Available from: https://docs.celeryq.dev/en/v5.3.6/getting-started/introduction.html

16. Welcome to Radon's documentation! — Radon 4.1.0 documentation [Internet]. [cited 2024 Apr 17]. Available from: https://radon.readthedocs.io/en/latest/

17. Ventresque A. Lecture 3: Code Quality [Internet]. Available from: https://tcd.blackboard.com/

18. Atik M. muatik/flask-profiler [Internet]. 2024 [cited 2024 Apr 17]. Available from: https://github.com/muatik/flask-profiler

19. memory-profiler · PyPI [Internet]. [cited 2024 Apr 17]. Available from: https://pypi.org/project/memory-profiler/

20. Locust - A modern load testing framework [Internet]. [cited 2024 Apr 17]. Available from: https://locust.io/

21. Coverage.py — Coverage.py 7.4.4 documentation [Internet]. [cited 2024 Apr 17]. Available from: https://coverage.readthedocs.io/en/7.4.4/index.html

22. Software testing metrics with formulas to measure QA performance [Internet]. [cited 2024 Apr 17]. Available from: https://zebrunner.com/blog-posts/65-testing-metrics

23. How do you deal with test case duplication, redundancy, and obsolescence? [Internet]. [cited 2024 Apr 17]. Available from: https://www.linkedin.com/advice/3/how-do-you-deal-test-case-duplication-redundancy

24. Full pytest documentation — pytest documentation [Internet]. [cited 2024 Apr 17]. Available from: https://docs.pytest.org/en/7.1.x/contents.html

25. Ghimire D. Comparative study on Python web frameworks: Flask and Django [Internet]. Metropolia University of Applied Sciences; 2020. Available from: https://www.theseus.fi/bitstream/handle/10024/339796/Ghimire_Devndra.pdf?sequence=2

26. asgiref · PyPI [Internet]. [cited 2024 Apr 17]. Available from: https://pypi.org/project/asgiref/

27. greenlet · PyPI [Internet]. [cited 2024 Apr 17]. Available from: https://pypi.org/project/greenlet/

28. Kumar S. python-dotenv: Read key-value pairs from a .env file and set them as environment variables [Internet]. [cited 2024 Apr 17]. Available from: https://github.com/theskumar/python-dotenv

29. GitHub [Internet]. [cited 2024 Apr 17]. Raise error with `pytest==8.0.0` · Issue #65 · TvoroG/pytest-lazy-fixture. Available from: https://github.com/TvoroG/pytest-lazy-fixture/issues/65

30. Software Engineering Intelligence | Code Climate [Internet]. [cited 2024 Apr 17]. Available from: https://codeclimate.com/