1. **Introduction**

This documentation outlines the architecture and security practices implemented in our Flask application, which seamlessly integrates with the HAProxy server.

The Flask application serves as the backbone of our system, responsible for orchestrating API calls to the HAProxy Data Plane API, managing data for frontend pages, and delivering a seamless user experience. To fortify our application against potential threats, we have implemented a comprehensive security strategy, with a specific focus on safeguarding the communication between the frontend and backend.

**2. High-Level Architecture**

HA Proxy LB

haproxy.cfg (main config file)

Config File deployment

Data Plane API

/tmp/haproxy.cfg

API Call

API Call

* Templates
* Static
* App
* Systemctl Service
* HA Proxy Config archivals

Flask Application

**3. Components**

**3.1. Frontend**

In Flask, templates are used to generate dynamic HTML content. Templates allow you to embed Python code within HTML files, making it easy to generate content based on data from your application. Jinja2 is the default templating engine used in Flask.

Here in this application HTML along with Javascript is used to generate dynamic content of a page according to the current configuration of HA proxy config file. Templates are rendered on flask server using main python file.

**3.2. Backend**

**3.2.1. APP**

The Flask App serves as the core backend component, responsible for two primary functions:

**API Calls to HAProxy Data Plane API:**

The Flask App is designed to facilitate API calls to the HAProxy Data Plane API. This includes handling requests to configure and manage the HAProxy data plane, allowing dynamic adjustments to the load balancing and routing settings.

These API calls are crucial for updating configurations, adding or removing backend servers, and ensuring the smooth operation of the HAProxy service.

**Data Handling for Frontend Pages:**

Additionally, the Flask App manages the data that is presented on the frontend pages. It interacts with the database or external services to retrieve relevant information, which is then formatted and sent to the frontend for display.

This includes processing user requests, querying data from storage, and organizing the information in a format suitable for rendering on the frontend.

By combining these functionalities, the Flask App acts as a bridge between the frontend and the HAProxy Data Plane API, ensuring seamless communication and interaction between the user interface and the underlying infrastructure. The modular design of the Flask App allows for flexibility and extensibility, supporting both API-related tasks and data manipulation for an efficient and responsive backend.

**4. Data Plane API Endpoints**

1. http://{server\_ip}/v2/services/haproxy/configuration/version [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#get-/services/haproxy/configuration/raw)]
2. http://{server\_ip}/v2/services/haproxy/transactions [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--Transactions)]
3. http://{server\_ip}/v2/services/haproxy/configuration/named\_defaults [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#get-/services/haproxy/configuration/named_defaults)]
4. http://{server\_ip}/v2/services/haproxy/configuration/global [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#get-/services/haproxy/configuration/global)]
5. http://{server\_ip}/v2/services/haproxy/configuration/backends [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--Backend)]
6. http://{server\_ip}/v2/services/haproxy/configuration/servers [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--Server)]
7. http://{server\_ip}/v2/services/haproxy/configuration/frontends [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--Frontend)]
8. http://{server\_ip}/v2/services/haproxy/configuration/binds [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--Bind)]
9. http://{server\_ip}/v2/services/haproxy/configuration/backend\_switching\_rules [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--BackendSwitchingRule)]
10. http://{server\_ip}/v2/services/haproxy/configuration/acls [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--ACL)]
11. http://{server\_ip}/v2/services/haproxy/configuration/raw [[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#get-/services/haproxy/configuration/raw)]
12. http://{server\_ip}/v2/services/haproxy/configuration/http\_request\_rules

[[Reference](https://www.haproxy.com/documentation/dataplaneapi/enterprise/#tag--HTTPRequestRule)]

**5. Authentication and Authorization**

**5.1. Flask Service Configuration:**

The Flask service is configured to run as the root user, granting it elevated privileges. This enables the Flask service to make changes to the HAProxy service seamlessly.

**5.2. Access Control for Flask App:**

Root or Sudo Users:

Only users with root or sudo privileges can make changes to the HA proxy configuration file.

Regular Users:

Regular users are limited to viewing the configuration set by the root or sudo user. Their access is restricted to read-only actions on frontend.

**5.3. Frontend Authentication:**

Accessing the frontend requires authentication using the server’s username and password. This authentication is validated at the operating system level. Only users with valid credentials are granted access to the Flask frontend.

**5.4. Data Plane API Authentication:**

For the data plane API, authentication credentials are embedded in the configuration file of the Flask app. This ensures secure access to the API, and only those with the proper credentials can interact with the data plane.

This structure emphasizes the separation of user roles and access levels, making it clear who has the authority to make changes to the Flask app, view configurations, access the frontend, and interact with the data plane API.

**6. Deployment**

The deployment of the Flask app is orchestrated as a service on the HAProxy server, ensuring a robust and continuous execution environment. The following steps outline the deployment process:

**Service Configuration:**

The Flask app is configured to run as a service on the HAProxy server. This involves creating a service file within the systemctl folder, specifying the settings and parameters required for the service.

**Systemctl Service File:**

A service file, typically located in the /etc/systemd/system/ directory, is crafted to define the Flask app as a service. This file includes details such as the executable path, environment variables, startup parameters, and dependencies.

Example Service File (/etc/systemd/system/flask\_haproxy.service):

Once the service file is configured, it needs to be enabled and started using systemctl commands:

sudo systemctl enable flask\_haproxy.service

sudo systemctl start flask\_haproxy.service

sudo systemctl status flask\_haproxy.service

Enabling the service ensures that it starts automatically upon system boot, and the start command initiates the service.

**Monitoring and Maintenance:**

Regular monitoring and maintenance of the deployed Flask app service are essential. Logs generated by the service can be reviewed using journalctl to identify any issues or anomalies.

sudo journalctl -u flask\_haproxy.service

Additionally, HAProxy server configurations should be inspected to guarantee seamless integration and communication between the Flask app and HAProxy.

By following this deployment approach, the Flask app becomes a managed service on the HAProxy server, benefiting from the system's supervision and ensuring continuous availability. Adjustments to the service configuration or updates can be easily applied and monitored through the systemctl interface.

**7. Security**

To bolster the security of the application, each request from the frontend to the backend is meticulously managed using CSRF (Cross-Site Request Forgery) tokens. This ensures protection against unauthorized and malicious actions. The CSRF token mechanism is employed with the following considerations:

**CSRF Token Generation:**

When a user logs into the frontend, a unique CSRF token is generated. This token serves as a cryptographic token tied to the user's session.

**Token Inclusion in Requests:**

Every subsequent request made from the frontend to the backend includes the CSRF token. This is typically achieved by adding the token to the request headers or embedding it within the request payload.

**Token Validation in Backend:**

Upon receiving a request, the Flask backend validates the CSRF token associated with the user's session. If the token is missing, incorrect, or expired, the request is considered unauthorized, and appropriate measures are taken.

**Token Expiry and Renewal:**

CSRF tokens have a finite lifespan to prevent long-term misuse. The backend manages token expiry, and the frontend is responsible for renewing the token as needed during user interactions.

**Session Management:**

The backend maintains secure user sessions, and the CSRF token is intricately linked to these sessions. If a user logs out or their session expires, the associated CSRF token becomes invalid.

**Protection Against CSRF Attacks:**

The CSRF token mechanism provides robust protection against Cross-Site Request Forgery attacks, ensuring that only authenticated and authorized requests are processed by the backend.

By enforcing the use of CSRF tokens, the application fortifies itself against unauthorized actions initiated by malicious entities. This security measure plays a pivotal role in maintaining the integrity and confidentiality of user interactions and data within the Flask application.