End-to-End Trace ID Implementation

This blog post was written with the assistance of ChatGPT-4o.

I worked on an end-to-end trace ID solution to ensure that every request and response in our system can be tracked consistently across the frontend and backend. This solution helps in debugging, monitoring, and logging by associating every operation with a unique trace ID. Below is a detailed explanation of how the solution works, along with code examples.

How It Works

Frontend

The frontend part of this solution involves generating a trace ID for each request and sending it along with client information to the backend. This trace ID is used to track the request through various stages of processing on the backend.

1. **Client Information Gathering**: We collect relevant information from the client, such as screen dimensions, network type, timezone, and more. This information is sent along with the request headers.
2. **Trace ID Generation**: A unique trace ID is generated for each request. This trace ID is included in the request headers, allowing us to trace the request through its lifecycle.
3. **API Fetch**: The `apiFetch` function is used to make API calls. It includes the trace ID and client information in the headers of each request.

Backend

The backend part of the solution involves logging the trace ID with each log message and including the trace ID in responses. This allows us to trace requests through the backend processing and match responses to requests.

1. **Trace ID Handling**: The backend receives the trace ID from the request headers or generates a new one if it is not provided. The trace ID is stored in a Flask global object for use throughout the request lifecycle.
2. **Logging**: Custom log formatters are used to include the trace ID in each log message. This ensures that all log messages related to a request can be correlated using the trace ID.
3. **Response Handling**: The trace ID is included in the response headers. If an error occurs, the trace ID is also included in the error response body to help with debugging.

Kibana

Kibana is a powerful tool for visualizing and searching log data stored in Elasticsearch. With our Trace ID solution, you can easily track and debug requests using Kibana. The trace ID, which is included in every log entry, can be used to filter and search for specific logs.

To search for logs with a specific trace ID, you can use the Kibana Query Language (KQL). For example, you can search for all logs related to a particular trace ID with the following query:

```
trace_id:"Lc6t"
```

This query will return all log entries that contain the trace ID "Lc6t," allowing you to trace the request's path through the system. Additionally, you can combine this query with other criteria to narrow down the search results, such as filtering by log level, timestamp, or specific keywords within the log messages.

By leveraging Kibana's visualization capabilities, you can also create dashboards that display metrics and trends based on the trace IDs. For instance, you can visualize the number of requests processed, average response times, and error rates, all correlated with their respective trace IDs. This helps in identifying patterns and potential issues in your application's performance and reliability.

Using Kibana in conjunction with our Trace ID solution provides a comprehensive approach to monitoring, debugging, and analyzing your system's behavior, ensuring that every request can be effectively tracked and investigated.

Frontend

api.js

```
const BASE_URL = process.env.REACT_APP_BASE_URL;

// Function to get client information
const getClientInfo = () => {
    const { language, platform, cookieEnabled, doNotTrack, onLine } = navigator;
    const { width, height } = window.screen;
    const connection = navigator.connection || navigator.mozConnection || navigator.webkitConnection;
    const networkType = connection ? connection.effectiveType : 'unknown';
    const timeZone = Intl.DateTimeFormat().resolvedOptions().timeZone;
    const referrer = document.referrer;
    const viewportWidth = window.innerWidth;
    const viewportHeight = window.innerHeight;
```

```
    return {
        screenWidth: width,
        screenHeight: height,
        networkType,
        timeZone,
        language,
        platform,
        cookieEnabled,
        doNotTrack,
        onLine,
        referrer,
        viewportWidth,
        viewportHeight
    };
};

// Function to generate a unique trace ID
export const generateTraceId = (length = 4) => {
    const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
    let traceId = '';
    for (let i = 0; i < length; i++) {
        const randomIndex = Math.floor(Math.random() * characters.length);
        traceId += characters.charAt(randomIndex);
    }
    return traceId;
};

export const apiFetch = async (endpoint, options = {}) => {
    const url = `${BASE_URL}${endpoint}`;
    const clientInfo = getClientInfo();

    const traceId = options.traceId || generateTraceId();

    const headers = {
        'Content-Type': 'application/json',
        'X-Client-Info': JSON.stringify(clientInfo),
```

```
        'X-Trace-Id': traceId,
        ...(options.headers || {})
    };

    const response = await fetch(url, {
        ...options,
        headers
    });

    return response;
};
```

App.js

```
try {
  const response = await apiFetch('api', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(content),
    traceId: traceId
  });

  if (response.ok) {
    const data = await response.json();
    //...
  } else {
    const errorData = await response.json();
    const errorMessage = errorData.message || 'An unknown error occurred';
    let errorToastMessage = errorMessage;
    errorToastMessage += ` (Trace ID: ${traceId})`;
    toast.error(errorToastMessage, {
      autoClose: 8000
    });
    setError(errorToastMessage);
  }
```

```
} catch (error) {
  let errorString = error instanceof Error ? error.message : JSON.stringify(error);

  const duration = (Date.now() - startTime) / 1000;

  if (error.response) {
    // The request was made and the server responded with a status code that falls out of the range of
    errorString += ` (HTTP ${error.response.status}: ${error.response.statusText})`;
    console.error('Response error data:', error.response.data);
  } else if (error.request) {
    // The request was made but no response was received
    errorString += ' (No response received)';
    console.error('Request error data:', error.request);
  } else {
    // Something happened in setting up the request that triggered an Error
    errorString += ` (Error setting up request: ${error.message})`;
  }

  errorString += ` (Trace ID: ${traceId})`;

  if (error instanceof Error) {
    errorString += `\nStack: ${error.stack}`;
  }

  errorString += JSON.stringify(error);

  errorString += ` (Duration: ${duration} seconds)`;

  toast.error(`Error: ${errorString}`, {
    autoClose: 8000
  });
  setError(errorString);
} finally {
  toast.dismiss(toastId);
}
```

Backend

__init__.py

```
# -*- encoding: utf-8 -*-

import os
import json
import time
import uuid
import string
import random

from flask import Flask, request, Response, g, has_request_context
from flask_cors import CORS

from .routes import initialize_routes
from .models import db, insert_default_config
import logging
from logging.handlers import RotatingFileHandler
from prometheus_client import Counter, generate_latest, Gauge
from flask_migrate import Migrate
from logstash_formatter import LogstashFormatterV1

app = Flask(__name__)

app.config.from_object('api.config.BaseConfig')

db.init_app(app)
initialize_routes(app)

CORS(app)

migrate = Migrate(app, db)


class RequestFormatter(logging.Formatter):
```

```
    def format(self, record):
        if has_request_context():
            record.trace_id = getattr(g, 'trace_id', 'unknown')
        else:
            record.trace_id = 'unknown'
        return super().format(record)


class CustomLogstashFormatter(LogstashFormatterV1):
    def format(self, record):
        if has_request_context():
            record.trace_id = getattr(g, 'trace_id', 'unknown')
        else:
            record.trace_id = 'unknown'
        return super().format(record)


def setup_loggers():
    logstash_handler = RotatingFileHandler(
        'app.log', maxBytes=100000000, backupCount=1)
    logstash_handler.setLevel(logging.DEBUG)
    logstash_formatter = CustomLogstashFormatter()
    logstash_handler.setFormatter(logstash_formatter)

    txt_handler = RotatingFileHandler(
        'plain.log', maxBytes=100000000, backupCount=1)
    txt_handler.setLevel(logging.DEBUG)
    txt_formatter = RequestFormatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d] [trace_id: %(trace_id)s]')
    txt_handler.setFormatter(txt_formatter)

    root_logger = logging.getLogger()
    root_logger.setLevel(logging.DEBUG)
    root_logger.addHandler(logstash_handler)
    root_logger.addHandler(txt_handler)
```

```
    app.logger.addHandler(logstash_handler)
    app.logger.addHandler(txt_handler)

    werkzeug_logger = logging.getLogger('werkzeug')
    werkzeug_logger.setLevel(logging.DEBUG)
    werkzeug_logger.addHandler(logstash_handler)
    werkzeug_logger.addHandler(txt_handler)


setup_loggers()


def generate_trace_id(length=4):
    characters = string.ascii_letters + string.digits
    return ''.join(random.choice(characters) for _ in range(length))


@app.before_request
def before_request():
    request.start_time = time.time()
    trace_id = request.headers.get('X-Trace-Id', generate_trace_id())
    g.trace_id = trace_id

    client_info = request.headers.get('X-Client-Info')
    if client_info:
        try:
            client_info_json = json.loads(client_info)
            logging.info(f"Client Info: {client_info_json}")
        except json.JSONDecodeError:
            logging.warning("Invalid JSON format for X-Client-Info header")


@app.after_request
def after_request(response):
    response.headers['X-Trace-Id'] = g.trace_id
```

```
    if response.status_code != 200:
        logging.error(f'Response status code: {response.status_code}')
        logging.error(f'Response body: {response.get_data(as_text=True)}')

        if response.content_type == 'application/json':
            try:
                response_json = response.get_json()
                response_json['trace_id'] = g.trace_id
                response.set_data(json.dumps(response_json))
            except Exception as e:
                logging.error(f"Error adding trace_id to response: {e}")

    return response
```

Log

You can search for all logs related to a particular trace ID with the following query:

```
trace_id:"Lc6t"
```


```
{
  "_index": "flask-logs-2024.07.05",
  "_type": "_doc",
  "_id": "Ae9zgZABqOMSOpxCZC5X",
  "_version": 1,
  "_score": 1,
  "_source": {
    "tags": [
      "_grokparsefailure"
    ],
    "filename": "generate.py",
    "funcName": "post",
    "message": "Request processed successfully",
    "@version": 1,
    "name": "root",
    "host": "ip-172-31-35-xxx.ec2.internal",
    "relativeCreated": 685817.8744316101,
    "levelname": "INFO",
```

```
    "created": 1720158740.894831,
    "thread": 139715118360128,
    "threadName": "Thread-5",
    "levelno": 20,
    "pathname": "/home/project/project-name/api/routes/generate.py",
    "msecs": 894.8309421539307,
    "processName": "MainProcess",
    "lineno": 287,
    "path": "/home/project/project-name/app.log",
    "args": [],
    "source_host": "ip-172-31-35-xxx.ec2.internal",
    "module": "generate",
    "trace_id": "Lc6t",
    "stack_info": null,
    "process": 107613,
    "@timestamp": "2024-07-05T05:52:20.894Z"
  },
  "fields": {
    "levelname.keyword": [
      "INFO"
    ],
    "tags.keyword": [
      "_grokparsefailure"
    ],
    "relativeCreated": [
      685817.9
    ],
    "processName.keyword": [
      "MainProcess"
    ],
    "filename.keyword": [
      "generate.py"
    ],
    "funcName": [
      "post"
    ],
```

```
"path": [
  "/home/project/project-name/app.log"
],
"processName": [
  "MainProcess"
],
"@version": [
  1
],
"host": [
  "ip-172-31-35-xxx.ec2.internal"
],
"msecs": [
  894.83093
],
"source_host.keyword": [
  "ip-172-31-35-xxx.ec2.internal"
],
"host.keyword": [
  "ip-172-31-35-xxx.ec2.internal"
],
"levelname": [
  "INFO"
],
"process": [
  107613
],
"threadName.keyword": [
  "Thread-5"
],
"trace_id": [
  "Lc6t"
],
"source_host": [
  "ip-172-31-35-xxx.ec2.internal"
],
```

```
"created": [
  1720158700
],
"module": [
  "generate"
],
"module.keyword": [
  "generate"
],
"name.keyword": [
  "root"
],
"thread": [
  139715118360128
],
"message": [
  "Request processed successfully"
],
"levelno": [
  20
],
"trace_id.keyword": [
  "Lc6t"
],
"threadName": [
  "Thread-5"
],
"pathname": [
  "/home/project/project-name/api/routes/generate.py"
],
"tags": [
  "_grokparsefailure"
],
"pathname.keyword": [
  "/home/project/project-name/api/routes/generate.py"
],
```

```
    "@timestamp": [
      "2024-07-05T05:52:20.894Z"
    ],
    "filename": [
      "generate.py"
    ],
    "lineno": [
      287
    ],
    "message.keyword": [
      "Request processed successfully"
    ],
    "name": [
      "root"
    ],
    "funcName.keyword": [
      "post"
    ],
    "path.keyword": [
      "/home/project/project-name/app.log"
    ]
  }
}
```


As shown above, you can see the trace ID in the log.