End-to-End Trace-ID-Implementierung

Dieser Blogbeitrag wurde mit Unterstützung von ChatGPT-4o verfasst.

Ich habe an einer End-to-End-Trace-ID-Lösung gearbeitet, um sicherzustellen, dass jede Anfrage und jede Antwort in unserem System konsistent über das Frontend und das Backend hinweg verfolgt werden kann. Diese Lösung unterstützt das Debugging, die Überwachung und die Protokollierung, indem jeder Vorgang mit einer eindeutigen Trace-ID verknüpft wird. Im Folgenden finden Sie eine detaillierte Erklärung, wie die Lösung funktioniert, zusammen mit Codebeispielen.

Wie es funktioniert

Frontend

Der Frontend-Teil dieser Lösung umfasst die Generierung einer Trace-ID für jede Anfrage und das Senden dieser zusammen mit Client-Informationen an das Backend. Diese Trace-ID wird verwendet, um die Anfrage durch verschiedene Verarbeitungsstufen im Backend zu verfolgen.

1. Erfassung von Client-Informationen: Wir sammeln relevante Informationen vom Client, wie Bildschirmgröße, Netzwerktyp, Zeitzone und mehr. Diese Informationen werden zusammen mit den Anfrage-Headern gesendet.
2. Trace-ID-Generierung: Für jede Anfrage wird eine eindeutige Trace-ID generiert. Diese Trace-ID wird in den Anfrage-Headern enthalten sein, wodurch wir die Anfrage während ihres gesamten Lebenszyklus verfolgen können.
3. API Fetch: Die Funktion `apiFetch` wird verwendet, um API-Aufrufe durchzuführen. Sie fügt die Trace-ID und Client-Informationen in den Headern jeder Anfrage hinzu.

Backend

Der Backend-Teil der Lösung beinhaltet das Protokollieren der Trace-ID mit jeder Log-Nachricht und das Einbeziehen der Trace-ID in die Antworten. Dies ermöglicht es uns, Anfragen durch die Backend-Verarbeitung zu verfolgen und Antworten den Anfragen zuzuordnen.

1. Trace-ID-Verarbeitung: Das Backend empfängt die Trace-ID aus den Anfrage-Headern oder generiert eine neue, falls keine bereitgestellt wird. Die Trace-ID wird in einem Flask-Global-Objekt gespeichert, um sie während des gesamten Anfragelebenszyklus zu verwenden.
2. Protokollierung: Benutzerdefinierte Log-Formate werden verwendet, um die Trace-ID in jede Log-Nachricht einzufügen. Dadurch wird sichergestellt, dass alle Log-Nachrichten, die zu einer Anfrage gehören, mithilfe der Trace-ID korreliert werden können.
3. Antwortverarbeitung: Die Trace-ID wird in den Antwort-Headern enthalten sein. Tritt ein Fehler auf, wird die Trace-ID auch im Fehlerantwort-Body enthalten sein, um das Debugging zu erleichtern.

Kibana

Kibana ist ein leistungsstarkes Tool zur Visualisierung und Suche von Log-Daten, die in Elasticsearch gespeichert sind. Mit unserer Trace-ID-Lösung können Sie Anfragen einfach mit Kibana verfolgen und debuggen. Die Trace-ID, die in jedem Log-Eintrag enthalten ist, kann verwendet werden, um spezifische Logs zu filtern und zu durchsuchen.

Um nach Protokollen mit einer bestimmten Trace-ID zu suchen, können Sie die Kibana Query Language (KQL) verwenden. Beispielsweise können Sie nach allen Protokollen suchen, die mit einer bestimmten Trace-ID verknüpft sind, indem Sie die folgende Abfrage verwenden:

```
trace_id:"Lc6t"
```

Diese Abfrage gibt alle Log-Einträge zurück, die die Trace-ID “Lc6t” enthalten, sodass Sie den Pfad der Anfrage durch das System nachverfolgen können. Darüber hinaus können Sie diese Abfrage mit anderen Kriterien kombinieren, um die Suchergebnisse einzugrenzen, z. B. durch Filtern nach Log-Level, Zeitstempel oder spezifischen Schlüsselwörtern innerhalb der Log-Nachrichten.

Durch die Nutzung der Visualisierungsfunktionen von Kibana können Sie auch Dashboards erstellen, die Metriken und Trends basierend auf den Trace-IDs anzeigen. Zum Beispiel können Sie die Anzahl der verarbeiteten Anfragen, die durchschnittlichen Antwortzeiten und die Fehlerraten visualisieren, die alle mit ihren jeweiligen Trace-IDs korreliert sind. Dies hilft dabei, Muster und potenzielle Probleme in der Leistung und Zuverlässigkeit Ihrer Anwendung zu identifizieren.

Die Verwendung von Kibana in Verbindung mit unserer Trace-ID-Lösung bietet einen umfassenden Ansatz zur Überwachung, Fehlerbehebung und Analyse des Systemverhaltens, wodurch sichergestellt wird, dass jede Anfrage effektiv nachverfolgt und untersucht werden kann.

Frontend

api.js

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

// Funktion zum Abrufen von Client-Informationen 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
};

};
```

// Funktion zur Generierung einer eindeutigen Trace-ID export const generateTraceId = (length = 4) => { const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu-

vwxyz0123456789’; 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 || 'Ein unbekannter Fehler ist aufgetreten';
    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) { // Die Anfrage wurde gemacht und der Server antwortete mit einem Statuscode, der außerhalb des Bereichs von 2xx liegt errorString += `(HTTP ${error.response.status}: ${error.response.statusText})`; console.error('Antwort-Fehlerdaten:', error.response.data); } else if (error.request) { // Die Anfrage wurde gemacht, aber es wurde keine Antwort empfangen errorString += ' (Keine Antwort empfangen)'; console.error('Anfrage-Fehlerdaten:', error.request); } else { // Beim Einrichten der Anfrage ist etwas passiert, das einen Fehler ausgelöst hat errorString += `(Fehler beim Einrichten der Anfrage: ${error.message})`; }

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

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

errorString += JSON.stringify(error);

errorString += `(Dauer: ${duration} Sekunden)`;

```
toast.error(`Fehler: ${errorString}`, {
    autoClose: 8000
  });
```

```
  setError(errorString);
} finally {
  toast.dismiss(toastId);
}
```

Backend

__init__.py

```
# -*- Kodierung: 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
```

Dieser Code importiert verschiedene Module und Funktionen, die für die Konfiguration und den Betrieb einer Flask-Anwendung benötigt werden. Hier ist eine kurze Erklärung der einzelnen Importe:

- `initialize_routes`: Eine Funktion, die die Routen der Anwendung initialisiert.
- `db` und `insert_default_config`: Datenbankmodelle und eine Funktion zum Einfügen von Standardkonfigurationen.
- `logging`: Das Python-Modul für die Protokollierung.

- `RotatingFileHandler`: Ein Handler für die Protokollierung, der die Logdateien rotiert, um Speicherplatz zu sparen.
- `Counter`, `generate_latest`, `Gauge`: Funktionen und Klassen aus der `prometheus_client`-Bibliothek zur Überwachung und Metrikenerfassung.
- `Migrate`: Eine Erweiterung für Flask, die Datenbankmigrationen verwaltet.
- `LogstashFormatterV1`: Ein Formatter für die Protokollierung, der Logs im Logstash-Format ausgibt.

```
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)
```

Diese Klasse `RequestFormatter` erweitert die `logging.Formatter`-Klasse und überschreibt die `format`-Methode. Wenn ein Request-Kontext vorhanden ist, wird die `trace_id` aus dem `g`-Objekt geholt und dem Logging-Record hinzugefügt. Falls kein Request-Kontext vorhanden ist, wird die `trace_id` auf `'unknown'` gesetzt. Schließlich wird die ursprüngliche `format`-Methode der übergeordneten Klasse aufgerufen, um den formatierten Log-Eintrag zurückzugeben.

```
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)
```

Übersetzung:

```
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)
```

In diesem Code wird eine benutzerdefinierte Klasse `CustomLogstashFormatter` erstellt, die von `LogstashFormatterV1` erbt. Die Methode `format` wird überschrieben, um sicherzustellen, dass der `trace_id`-Wert im Log-Datensatz gesetzt wird. Wenn ein Request-Kontext vorhanden ist, wird die `trace_id` aus dem globalen Objekt `g` geholt. Falls kein Request-Kontext vorhanden ist, wird der Wert auf `'unknown'` gesetzt. Schließlich wird die ursprüngliche `format`-Methode der Elternklasse aufgerufen, um den formatierten Log-Datensatz zurückzugeben.

```
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))
```

Übersetzung:

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

Die Funktion `generate_trace_id` generiert eine zufällige Zeichenkette (Trace-ID) mit einer Standardlänge von 4 Zeichen. Die Zeichenkette besteht aus einer Kombination von Buchstaben (Groß- und Kleinbuchstaben) und Ziffern. Die Länge der Zeichenkette kann durch den Parameter `length` angepasst werden.

```
@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("Ungültiges JSON-Format für den 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'Antwortstatuscode: {response.status_code}')
    logging.error(f'Antworttext: {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"Fehler beim Hinzufügen der trace_id zur Antwort: {e}")

return response

## Protokoll

Sie können nach allen Protokollen suchen, die mit einer bestimmten Trace-ID verknüpft sind, indem Sie di
```

trace_id:“Lc6t”

```
(Anmerkung: Der Code-Block bleibt unverändert, da es sich um eine technische ID handelt, die nicht übers

```json
{
  "_index": "flask-logs-2024.07.05",
  "_type": "_doc",
  "_id": "Ae9zgZABqOMSOpxCZC5X",
  "_version": 1,
  "_score": 1,
  "_source": {
    "tags": [
      "_grokparsefailure"
    ],
    "filename": "generate.py",
    "funcName": "post",
    "message": "Anfrage erfolgreich verarbeitet",
    "@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": [
  "Anfrage erfolgreich verarbeitet"
],
"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": [
      "Anfrage erfolgreich verarbeitet"
    ],
    "name": [
      "root"
    ],
    "funcName.keyword": [
      "post"
    ],
    "path.keyword": [
      "/home/project/project-name/app.log"
    ]
  }
}
```


Wie oben gezeigt, können Sie die Trace-ID im Protokoll sehen.