एंड-टू-एंड ट्रेस आईडी कार्यान्वयन

यह ब्लॉग पोस्ट □□□□□□□-4□ की सहायता से लिखा गया है।

मैंने एक एंड-टू-एंड ट्रेस □□ समाधान पर काम किया है ताकि हमारे सिस्टम में हर अनुरोध और प्रतिक्रिया को फ्रंटएंड और बैकएंड में सुसंगत रूप से ट्रैक किया जा सके। यह समाधान डीबगिंग, मॉनिटरिंग और लॉगिंग में मदद करता है, क्योंकि यह हर ऑपरेशन को एक अद्वितीय ट्रेस □□ के साथ जोड़ता है। नीचे इस समाधान के काम करने का विस्तृत विवरण दिया गया है, साथ ही कोड उदाहरण भी दिए गए हैं।

यह कैसे काम करता है

फ्रंटएंड

इस समाधान के फ्रंटएंड भाग में प्रत्येक अनुरोध के लिए एक ट्रेस □□ उत्पन्न करना और इसे क्लाइंट जानकारी के साथ बैकएंड पर भेजना शामिल है। यह ट्रेस □□ बैकएंड पर प्रसंस्करण के विभिन्न चरणों के माध्यम से अनुरोध को ट्रैक करने के लिए उपयोग किया जाता है।

1. क्लाइंट जानकारी एकत्र करना: हम क्लाइंट से संबंधित जानकारी एकत्र करते हैं, जैसे स्क्रीन का आयाम, नेटवर्क प्रकार, समय क्षेत्र, और अन्य। यह जानकारी अनुरोध हेडर के साथ भेजी जाती है।
2. ट्रेस □□ जनरेशन: प्रत्येक अनुरोध के लिए एक अद्वितीय ट्रेस □□ जनरेट की जाती है। यह ट्रेस □□ अनुरोध हेडर में शामिल होती है, जो हमें अनुरोध को उसके जीवनचक्र के माध्यम से ट्रेस करने की अनुमति देती है।
3. □□□ □□□□□: `apiFetch` फ़ंक्शन का उपयोग □□□ कॉल करने के लिए किया जाता है। यह प्रत्येक अनुरोध के हेडर में ट्रेस □□ और क्लाइंट जानकारी को शामिल करता है।

बैकएंड

समाधान के बैकएंड भाग में प्रत्येक लॉग संदेश के साथ ट्रेस □□ को लॉग करना और प्रतिक्रियाओं में ट्रेस □□ को शामिल करना शामिल है। यह हमें बैकएंड प्रसंस्करण के माध्यम से अनुरोधों को ट्रेस करने और अनुरोधों के साथ प्रतिक्रियाओं को मिलाने की अनुमति देता है।

1. ट्रेस □□ हैंडलिंग: बैकएंड अनुरोध हेडर से ट्रेस □□ प्राप्त करता है या यदि यह प्रदान नहीं किया गया है तो एक नया ट्रेस □□ जनरेट करता है। ट्रेस □□ को अनुरोध जीवनचक्र के दौरान उपयोग के लिए □□□□□ ग्लोबल ऑब्जेक्ट में संग्रहीत किया जाता है।
2. लॉगिंग: कस्टम लॉग फॉर्मेटर्स का उपयोग प्रत्येक लॉग संदेश में ट्रेस □□ को शामिल करने के लिए किया जाता है। यह सुनिश्चित करता है कि किसी अनुरोध से संबंधित सभी लॉग संदेशों को ट्रेस □□ का उपयोग करके सहसंबंधित किया जा सकता है।
3. प्रतिक्रिया प्रबंधन: ट्रेस □□ को प्रतिक्रिया हेडर में शामिल किया जाता है। यदि कोई त्रुटि होती है, तो ट्रेस □□ को डीबगिंग में सहायता के लिए त्रुटि प्रतिक्रिया बॉडी में भी शामिल किया जाता है।

□□□□□□□

□□□□□□□ □□□□□□□□□□□□□□□ में संग्रहीत लॉग डेटा को विज़ुअलाइज़ और खोजने के लिए एक शक्तिशाली टूल है। हमारे □□□□□ □□ समाधान के साथ, आप □□□□□□□ का उपयोग करके आसानी से अनुरोधों को ट्रैक और डीबग कर सकते हैं। ट्रेस □□, जो हर लॉग एंट्री में शामिल होता है, का उपयोग विशिष्ट लॉग्स को फ़िल्टर और खोजने के लिए किया जा सकता है।

किसी विशिष्ट ट्रेस □□ के साथ लॉग्स को खोजने के लिए, आप □□□□□□□ □□□□□ □□□□□□□□□ (□□□) का उपयोग कर सकते हैं। उदाहरण के लिए, आप किसी विशेष ट्रेस □□ से संबंधित सभी लॉग्स को निम्नलिखित क्वेरी के साथ खोज सकते हैं:

```
trace_id:"Lc6t"
```

यह क्वेरी सभी लॉग एंट्रीज़ को वापस करेगी जिनमें ट्रेस आईडी "□□6□" शामिल है, जिससे आप सिस्टम के माध्यम से अनुरोध के पथ का पता लगा सकते हैं। इसके अलावा, आप इस क्वेरी को अन्य मानदंडों के साथ जोड़कर खोज परिणामों को और संकीर्ण कर सकते हैं, जैसे कि लॉग स्तर, टाइमस्टैम्प, या लॉग संदेशों में विशिष्ट कीवर्ड के आधार पर फ़िल्टर करना।

□□□□□□□ की विज़ुअलाइज़ेशन क्षमताओं का उपयोग करके, आप डैशबोर्ड भी बना सकते हैं जो ट्रेस □□□ के आधार पर मेट्रिक्स और ट्रेंड्स प्रदर्शित करते हैं। उदाहरण के लिए, आप संसाधित अनुरोधों की संख्या, औसत प्रतिक्रिया समय और त्रुटि दरों को विज़ुअलाइज़ कर सकते हैं, जो उनके संबंधित ट्रेस □□□ के साथ सहसंबद्ध होते हैं। यह आपके एप्लिकेशन के प्रदर्शन और विश्वसनीयता में पैटर्न और संभावित समस्याओं की पहचान करने में मदद करता है।

हमारे □□□□□ □□ समाधान के साथ □□□□□□□ का उपयोग करने से आपके सिस्टम के व्यवहार की निगरानी, डिबगिंग और विश्लेषण करने के लिए एक व्यापक दृष्टिकोण प्रदान होता है, जिससे यह सुनिश्चित होता है कि हर अनुरोध को प्रभावी ढंग से ट्रैक और जांचा जा सकता है।

फ्रंटएंड

api.js

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

// क्लाइंट जानकारी प्राप्त करने के लिए फ़ंक्शन □□□□□ □□□□□□□□□□□□□□ = () => { □□□□□ { □□□□□□□□□, □□□□□, □□□□□□□□□□□□□□, □□□□□□□□□□□□, □□□□□□□ } = □□□□□□□□□□; □□□□□□ { □□□□□□, □□□□□□□ } = □□□□□□□.□□□□□□□; □□□□□□ □□□□□□□□□□□□ = □□□□□□□□□□.□□□□□□□□□□□□ || □□□□□□□□□□.□□□□□□□□□□□□□□□ || □□□□□□□□.□□□□□□□□□□□□□□□□□□□; □□□□□□ □□□□□□□□□□□□□ = □□□□□□□□□□□□ ? □□□□□□□□□□□□.□□□□□□□□□□□□□□□□ : '□□□□□□□□'; □□□□□□ □□□□□□□□□ = □□□□□.□□□□□□□□□□□□□□□□□().□□□□□□□□□□□□□□□□□().□□□□□□□□□; □□□□□□ □□□□□□□□□ = □□□□□□□□□.□□□□□□□□□; □□□□□□ □□□□□□□□□□□□□□□ = □□□□□□□.□□□□□□□□□□□□; □□□□□□ □□□□□□□□□□□□ = □□□□□□□.□□□□□□□□□□□□□;

return {
    screenWidth: width,
```

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

};

// एक अद्वितीय ट्रेस □□ जनरेट करने के लिए फ़ंक्शन □□□□□□ □□□□□ □□□□□□□□□□□□□□□ = (□□□□□□ = 4) => { □□□□□ □□□□□□□□□□ = '□□□0123456789'; □□□ □□□□□□ = ' '; □□□ (□□□ □ = 0; □ < □□□□□□; □++) { □□□□□ □□□□□□□□□□□ = □□□□.□□□□□(□□□□.□□□□□□() * □□□□□□□□□□.□□□□□□); □□□□□□ += □□□□□□□□□□.□□□□□□(□□□□□□□□□□□); } □□□□□□ □□□□□□; };

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

यह कोड एक `apiFetch` फ़ंक्शन को परिभाषित करता है जो एक एसिंक्रोनस (□□□□□) फ़ंक्शन है। यह फ़ंक्शन एक `endpoint` और वैकल्पिक `options` ऑब्जेक्ट को पैरामीटर के रूप में लेता है। फ़ंक्शन के अंदर, `BASE_URL` और `endpoint` को जोड़कर एक पूर्ण □□□ बनाया जाता है। इसके बाद, `getClientInfo()` फ़ंक्शन को कॉल करके क्लाइंट की जानकारी प्राप्त की जाती है।

```
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 || '           ';
    let errorToastMessage = errorMessage;
    errorToastMessage += ` (   ID: ${traceId})`;
    toast.error(errorToastMessage, {
      autoClose: 8000
    });
    setError(errorToastMessage);
}
} catch (error) {
  let errorString = error instanceof Error ? error.message : JSON.stringify(error);
```

□□□□□□ □□□□□□□□□□ = (□□□□□.□□□□() - □□□□□□□□□□□) / 1000;

```
  if (error.response) {
    //                    2xx
```

```
    errorString += ` (HTTP ${error.response.status}: ${error.response.statusText})`;
    console.error('           :', error.response.data);
  } else if (error.request) {
    //
    errorString += ' (                )';
    console.error('         :', error.request);
  } else {
    //
    errorString += ` (              : ${error.message})`;
  }

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

(नोट: यह कोड ब्लॉक है, इसलिए इसे अनुवादित नहीं किया गया है।)

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

हिंदी अनुवाद:

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

इस कोड में, यदि `error` एक `Error` ऑब्जेक्ट है, तो `errorString` में `error` का स्टैक ट्रेस जोड़ा जाता है।

```
□□□□□□□□□□□□ += □□□□.□□□□□□□□□□(□□□□□□);

□□□□□□□□□□□□ += (   : ${duration}   );

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

बैकएंड

__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
```

यह कोड □□□□□ वेब फ्रेमवर्क का उपयोग करता है और इसमें Flask, request, Response, g, और has_request_context को आयात किया गया है। साथ ही, flask_cors से CORS को भी आयात किया गया है। CORS (□□□□□-□□□□□□ □□□□□□□□ □□□□□□□) का उपयोग वेब एप्लिकेशन को अलग-अलग डोमेन से संसाधनों को साझा करने की अनुमति देने के लिए किया जाता है।

```
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
```

इस कोड को हिंदी में अनुवाद करने की आवश्यकता नहीं है क्योंकि यह □□□□□□ कोड है और इसे उसी रूप में रखा जाना चाहिए। यह कोड एक □□□□□ एप्लिकेशन में उपयोग होने वाले विभिन्न मॉड्यूल और लाइब्रेरीज़ को इम्पोर्ट करता है।

```
app = Flask(__name__)

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

यह कोड □□□□□ एप्लिकेशन के कॉन्फ़िगरेशन को api.config मॉड्यूल में परिभाषित BaseConfig क्लास से लोड करता है। इसका उपयोग एप्लिकेशन की सेटिंग्स और कॉन्फ़िगरेशन वैल्यूज़ को प्रबंधित करने के लिए किया जाता है।

```
db.init_app(app)
initialize_routes(app)
```

□□□□(□□□)

```
migrate = Migrate(app, db)
```

यह कोड □□□□□-□□□□□□□□ का उपयोग करके □□□□□ एप्लिकेशन और □□□□□□□□□□□ डेटाबेस के बीच माइग्रेशन को प्रबंधित करने के लिए है। `Migrate` क्लास `app` और `db` को इनिशियलाइज़ करती है, जिससे डेटाबेस स्कीमा में परिवर्तनों को आसानी से माइग्रेट किया जा सकता है।

```
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)
```

यह कोड एक `RequestFormatter` क्लास को परिभाषित करता है जो `logging.Formatter` को इन्हेरिट करता है। इस क्लास में `format` मेथड को ओवरराइड किया गया है। यह मेथड लॉग रिकॉर्ड को फॉर्मेट करने से पहले चेक करता है कि क्या रिक्वेस्ट कॉन्टेक्स्ट उपलब्ध है। यदि रिक्वेस्ट कॉन्टेक्स्ट उपलब्ध है, तो यह `trace_id` को `g` ऑब्जेक्ट से प्राप्त करता है और इसे लॉग रिकॉर्ड में जोड़ता है। यदि रिक्वेस्ट कॉन्टेक्स्ट उपलब्ध नहीं है, तो `trace_id` को '□□□□□□□□' के रूप में सेट किया जाता है। अंत में, यह मेथड पैरेंट क्लास के `format` मेथड को कॉल करता है और फॉर्मेटेड रिकॉर्ड को रिटर्न करता है।

```
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)
```

इस कोड में, `CustomLogstashFormatter` नामक एक क्लास है जो `LogstashFormatterV1` को इनहेरिट करती है। इस क्लास में `format` नामक एक मेथड है जो `record` नामक एक पैरामीटर लेती है।

1. `has_request_context()` फंक्शन का उपयोग करके यह जांचा जाता है कि क्या रिक्वेस्ट कॉन्टेक्स्ट उपलब्ध है।
2. यदि रिक्वेस्ट कॉन्टेक्स्ट उपलब्ध है, तो `record.trace_id` को `g` ऑब्जेक्ट से `trace_id` एट्रिब्यूट का मान दिया जाता है। यदि `trace_id` एट्रिब्यूट उपलब्ध नहीं है, तो डिफ़ॉल्ट मान `'unknown'` सेट किया जाता है।
3. यदि रिक्वेस्ट कॉन्टेक्स्ट उपलब्ध नहीं है, तो `record.trace_id` को `'unknown'` सेट किया जाता है।
4. अंत में, `super().format(record)` का उपयोग करके पैरेंट क्लास (`LogstashFormatterV1`) का `format` मेथड कॉल किया जाता है और उसका रिजल्ट रिटर्न किया जाता है।

```
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)
```

यह कोड एक `RotatingFileHandler` बनाता है जो लॉग फ़ाइल को रोटेट करता है, यानी जब फ़ाइल का आकार 100,000,000 बाइट्स (लगभग 100 □□) तक पहुंच जाता है, तो यह एक बैकअप फ़ाइल बनाता है और नई फ़ाइल में लॉगिंग शुरू करता है। `backupCount=1` का मतलब है कि केवल एक बैकअप फ़ाइल रखी जाएगी।

`txt_handler.setLevel(logging.DEBUG)` लॉगिंग लेवल को □□□□□ पर सेट करता है, जिसका अर्थ है कि सभी □□□□□ और उससे ऊपर के लॉग संदेश (□□□□, □□□□□□□, □□□□□, □□□□□□□□) लॉग किए जाएंगे।

`txt_formatter` एक कस्टम फॉर्मेटर है जो लॉग संदेशों को एक विशिष्ट प्रारूप में प्रदर्शित करता है। यह फॉर्मेटर समय (`asctime`), लॉग लेवल (`levelname`), संदेश (`message`), फ़ाइल का पथ (`pathname`), लाइन नंबर (`lineno`), और एक ट्रेस आईडी (`trace_id`) को शामिल करता है।

अंत में, `txt_handler.setFormatter(txt_formatter)` फॉर्मेटर को हैंडलर से जोड़ता है, ताकि लॉग संदेश उसी प्रारूप में लिखे जाएं।

```
    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)
```

यह कोड `werkzeug` नामक लॉगर को सेट करता है और उसे `DEBUG` लेवल पर कॉन्फ़िगर करता है। फिर यह `logstash_handler` और `txt_handler` दोनों हैंडलर्स को इस लॉगर में जोड़ता है। इसका मतलब है कि `werkzeug` से संबंधित सभी लॉग संदेश `DEBUG` लेवल और उससे ऊपर के, दोनों हैंडलर्स को भेजे जाएंगे।

```
setup_loggers()

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

यह फ़ंक्शन `generate_trace_id` एक यूनिक ट्रेस आईडी जनरेट करता है। यह आईडी `length` पैरामीटर द्वारा निर्धारित लंबाई की होती है, जिसका डिफ़ॉल्ट मान 4 है। आईडी बनाने के लिए यह `string.ascii_letters` और `string.digits` से यादृच्छिक वर्णों का चयन करता है और उन्हें जोड़कर एक स्ट्रिंग के रूप में लौटाता है।

```
@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_json}")
        except json.JSONDecodeError:
            logging.warning("X-Client-Info        JSON         ")

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

यह कोड □□□□□ एप्लिकेशन में एक `after_request` हुक को परिभाषित करता है। यह हुक हर □□□□ अनुरोध के बाद चलता है और प्रतिक्रिया (□□□□□□□□□) के हेडर में `X-Trace-Id` नामक एक हेडर जोड़ता है। इस हेडर का मान `g.trace_id` से लिया जाता है, जो □□□□□ के `g` ऑब्जेक्ट में संग्रहीत है। यह ट्रेस आईडी आमतौर पर अनुरोधों को ट्रैक करने के लिए उपयोग की जाती है।

```
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
```

लॉग

आप निम्नलिखित क्वेरी का उपयोग करके किसी विशेष ट्रेस □□ से संबंधित सभी लॉग्स को खोज सकते हैं:

```
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"
    ]
  }
}
```


जैसा कि ऊपर दिखाया गया है, आप लॉग में ट्रेस □□ देख सकते हैं।