エンドツーエンドのトレース ID 実装

このブログ記事は、ChatGPT-4o の助けを借りて書かれました。

私は、システム内のすべてのリクエストとレスポンスがフロントエンドとバックエンドで一貫して追跡できるようにするためのエンドツーエンドのトレース ID ソリューションに取り組みました。このソリューションは、すべての操作に一意のトレース ID を関連付けることで、デバッグ、監視、ロギングを支援します。以下に、このソリューションの動作の詳細な説明とコード例を示します。

仕組み

フロントエンド

このソリューションのフロントエンド部分では、各リクエストに対してトレース ID を生成し、それをクライアント情報と共にバックエンドに送信します。このトレース ID は、バックエンドでの処理のさまざまな段階を通じてリクエストを追跡するために使用されます。

1. クライアント情報の収集: 画面サイズ、ネットワークタイプ、タイムゾーンなど、クライアントから関連情報を収集します。この情報はリクエストヘッダーと共に送信されます。
2. トレース ID の生成: 各リクエストに対して一意のトレース ID が生成されます。このトレース ID はリクエストヘッダーに含まれ、リクエストのライフサイクルを通じて追跡できるようにします。
3. API フェッチ: `apiFetch` 関数は API 呼び出しを行うために使用されます。各リクエストのヘッダーにトレース ID とクライアント情報を含めます。

バックエンド

ソリューションのバックエンド部分では、各ログメッセージにトレース ID を記録し、レスポンスにトレース ID を含めることが含まれます。これにより、バックエンド処理を通じてリクエストを追跡し、レスポンスをリクエストにマッチさせることが可能になります。

1. Trace ID の処理: バックエンドはリクエストヘッダーから trace ID を受け取るか、提供されていない場合は新しいものを生成します。trace ID はリクエストライフサイクル全体で使用するために Flask のグローバルオブジェクトに保存されます。

2. ロギング: カスタムログフォーマッタを使用して、各ログメッセージにトレース ID を含めます。これにより、リクエストに関連するすべてのログメッセージをトレース ID を使用して関連付けることができます。
3. レスポンス処理: トレース ID はレスポンスヘッダーに含まれます。エラーが発生した場合、デバッグを支援するためにエラーレスポンスの本文にもトレース ID が含まれます。

Kibana

Kibana は、Elasticsearch に保存されたログデータを視覚化および検索するための強力なツールです。私たちの Trace ID ソリューションを使用すると、Kibana を使ってリクエストを簡単に追跡およびデバッグできます。すべてのログエントリに含まれるトレース ID を使用して、特定のログをフィルタリングおよび検索することができます。

特定のトレース ID に関連するログを検索するには、Kibana Query Language (KQL) を使用できます。例えば、特定のトレース ID に関連するすべてのログを検索するには、以下のクエリを使用します:

```
trace_id:"Lc6t"
```

このクエリは、トレース ID「Lc6t」を含むすべてのログエントリを返します。これにより、リクエストがシステム内をどのように通過したかを追跡することができます。さらに、このクエリを他の条件と組み合わせて、検索結果を絞り込むことも可能です。例えば、ログレベル、タイムスタンプ、またはログメッセージ内の特定のキーワードでフィルタリングすることができます。

Kibana の可視化機能を活用することで、トレース ID に基づいたメトリクスやトレンドを表示するダッシュボードを作成することもできます。例えば、処理されたリクエストの数、平均応答時間、エラーレートなどを、それぞれのトレース ID と関連付けて可視化することが可能です。これにより、アプリケーションのパフォーマンスと信頼性におけるパターンや潜在的な問題を特定するのに役立ちます。

Kibana を私たちの Trace ID ソリューションと組み合わせて使用することで、システムの動作を包括的に監視、デバッグ、分析するアプローチを提供し、すべてのリクエストを効果的に追跡および調査できるようになります。

フロントエンド

`api.js`

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

// クライアント情報を取得する関数 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
    };
};

// ユニークなトレース ID を生成する関数
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 || '不明なエラーが発生しました';
    let errorToastMessage = errorMessage;
    errorToastMessage += ` (トレース 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) {
    // リクエストが行われ、サーバーが 2xx の範囲外のステータスコードで応答した場合
    errorString += ` (HTTP ${error.response.status}: ${error.response.statusText})`;
    console.error('レスポンスエラーデータ:', error.response.data);
} else if (error.request) {
    // リクエストが行われたが、応答がなかった場合
    errorString += ' (応答がありませんでした)';
    console.error('リクエストエラーデータ:', error.request);
} else {
    // リクエストの設定中にエラーが発生した場合
    errorString += ` (リクエスト設定中のエラー: ${error.message})`;
}

errorString += ` (トレース ID: ${traceId})`;

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

このコードは、`error` が `Error` のインスタンスである場合に、エラーのスタックトレースを `errorString` に追加します。`error.stack` は、エラーが発生した際の呼び出しスタックを表す文字列です。これにより、エラーの発生源や経路を追跡するのに役立ちます。

```
errorString += JSON.stringify(error);
```

このコードは、`error` オブジェクトを JSON 形式の文字列に変換し、それを `errorString` という変数に追加しています。`JSON.stringify()` は、JavaScript オブジェクトを JSON 文字列に変換するための関数です。この行が実行されると、`error` オブジェクトの内容が文字列として `errorString` に追加されます。

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

このコードは、エラーメッセージに実行時間（秒単位）を追加するためのものです。`errorString` という文字列に、`Duration: X seconds` という形式で実行時間が追加されます。ここで、`${duration}` は変数 `duration` の値に置き換えられます。

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

バックエンド

__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
```

このコードは、Python の Flask アプリケーションで使用されるいくつかの重要なモジュールと関数をインポートしています。以下に各インポートの概要を説明します：

1. `initialize_routes`: アプリケーションのルートを初期化するための関数です。これにより、アプリケーションのエンドポイントが設定されます。
2. `db` と `insert_default_config`: データベースモデルとデフォルト設定を挿入するための関数です。`db` はデータベース接続を管理し、`insert_default_config` はアプリケーションの初期設定を行うために使用されます。
3. `logging`: Python の標準ロギングモジュールです。アプリケーションのログを記録するために使用されます。
4. `RotatingFileHandler`: ログファイルをローテーションするためのハンドラです。これにより、ログファイルが大きくなりすぎるのを防ぎます。
5. `Counter`, `generate_latest`, `Gauge`: Prometheus のメトリクスを収集するためのクラスと関数です。`Counter` はイベントの数をカウントし、`Gauge` は現在の値を記録します。`generate_latest` はメトリクスデータを生成します。
6. `Migrate`: Flask-Migrate のクラスで、データベースのマイグレーションを管理します。これにより、データベーススキーマの変更を簡単に適用できます。
7. `LogstashFormatterV1`: Logstash 形式のログをフォーマットするためのクラスです。これにより、ログデータを Logstash と互換性のある形式で出力できます。

これらのインポートは、Flask アプリケーションの基本的な機能を構築するために使用されます。

```
app = Flask(__name__)

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

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

このコードは、Flask アプリケーションでデータベースとルートを初期化するためのものです。`db.init_app(app)` は、Flask アプリケーションにデータベースを関連付け、`initialize_routes(app)` はアプリケーションのルート（エンドポイント）を設定します。

```
CORS(app)
```

このコードは、Flask アプリケーションで CORS（Cross-Origin Resource Sharing）を有効にするために使用されます。`CORS(app)` と記述することで、異なるオリジン（ドメイン、プロトコル、ポート）からのリクエストを許可することができます。これにより、フロントエンドとバックエンドが別々のドメインでホストされている場合でも、スムーズに通信を行うことが可能になります。

```
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)
```

このコードは、ログのフォーマットをカスタマイズするための `RequestFormatter` クラスを定義しています。このクラスは `logging.Formatter` を継承しており、`format` メソッドをオーバーライドしています。

- `has_request_context()` は、現在のスレッドがリクエストコンテキストを持っているかどうかをチェックします。
- リクエストコンテキストがある場合、`g.trace_id` を取得し、`record.trace_id` に設定します。`g.trace_id` が存在しない場合は、デフォルト値として `'unknown'` が使用されます。
- リクエストコンテキストがない場合、`record.trace_id` は `'unknown'` に設定されます。
- 最後に、親クラスの `format` メソッドを呼び出して、ログレコードをフォーマットします。

このフォーマッタを使用することで、ログにリクエストごとの `trace_id` を含めることができます。

```
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
```

このコードは、Flask アプリケーションでリクエストが処理される前に実行される before_request 関数を定義しています。具体的には、リクエストの開始時刻を記録し、リクエストヘッダーから X-Trace-Id を取得します。もし X-Trace-Id が存在しない場合は、新しいトレース ID を生成します。そして、そのトレース ID を Flask の g オブジェクトに保存します。これにより、リクエストの処理中にトレース ID を簡単に参照できるようになります。

```
    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("X-Client-Info ヘッダーの JSON 形式が無効です")

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

    if response.status_code != 200:
        logging.error(f'レスポンスステータスコード: {response.status_code}')
        logging.error(f'レスポンスボディ: {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" レスポンスに trace_id を追加する際にエラーが発生しました: {e}")

    return response
```

ログ

特定のトレース ID に関連するすべてのログを検索するには、次のクエリを使用できます:

```
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": " リクエストが正常に処理されました",
    "@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": [
  " リクエストが正常に処理されました"
],
```

```
"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": [
  " リクエストが正常に処理されました"
],
"name": [
  "root"
],
"funcName.keyword": [
  "post"
],
```

```
    "path.keyword": [
      "/home/project/project-name/app.log"
    ]
  }
}
```

上記のように、ログ内にトレース ID が表示されていることがわかります。