

2. 적용하기

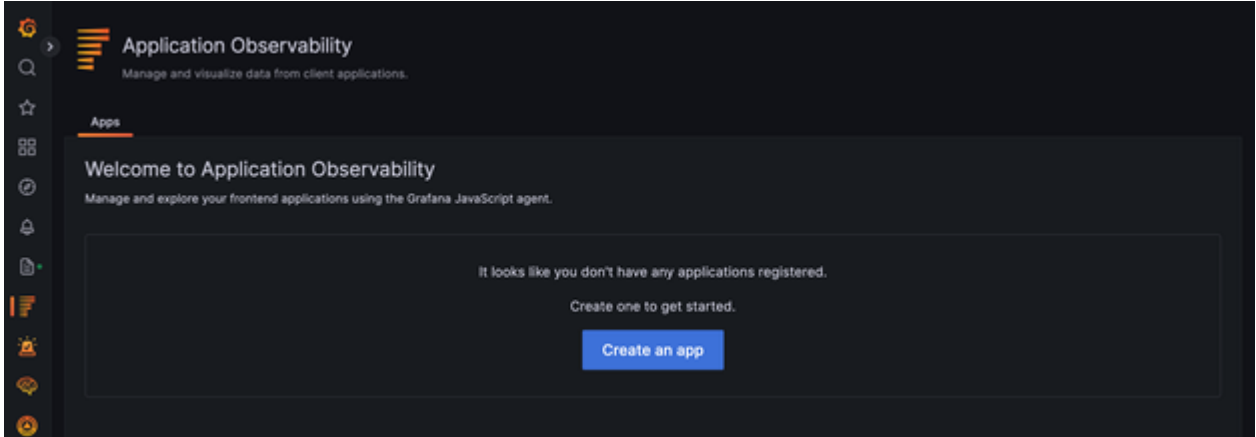
▼ Grafana Cloud에 Faro 설정하기




현재 Faro를 Grafana Cloud에서 사용하는 것은 private beta로 운영되고 있기에, 사전 신청을 한 후 승인이 되어야 사용할 수 있습니다. 따라서 이 글에서는 private beta 사용 승인이 완료되었다는 가정 하에 진행합니다. 대신 [이 문서](#)에 따라 Faro가 직접 구축한 Grafana Agent로 데이터를 보내도록 할 수 있습니다. 여기서는 Grafana Cloud를 사용합니다.

Frontend Application 생성하기

- 여기서 말하는 *Frontend Application*이란 Faro에서 생성할 프로젝트의 단위를 의미합니다. 아래 사진처럼 왼쪽에서 Faro의 로고를 클릭하면 앱 생성 페이지로 이동합니다.



- 그리고 아래와 같이 기본적인 정보를 입력해 애플리케이션을 생성합니다.



Application Observability

Manage and visualize data from client applications.

Apps

App Name

faro-demo-react

CORS Allowed Origins ?

Origin

localhost:8000

+ Add origin

Extra Log Labels

app

faro-demo-react

+ Add label

Create

Cancel

- 위 화면에서 CORS Allowed Origins 에 올바른 주소를 입력해야 React 애플리케이션이 데이터를 Faro에게 전달할 때 CORS error가 발생하지 않습니다.
- 생성을 완료하면 아래와 같이 초기 설정을 위한 페이지가 나타납니다.

faro-demo-react

App: faro-demo-react

Overview

Errors

Settings

Faro Web SDK Configuration

✓

App created successfully!

Faro Web SDK Configuration

To start capturing telemetry for your application, include the following snippet in your JavaScript Initialization code. Go check out our [documentation](#) for advanced use cases.

1. Initialize Faro Web SDK

2. Example usage

NPM

CDN

```
import { initializeFaro } from '@grafana/faro-web-sdk';

const faro = initializeFaro({
  url: 'https://faro-collector-prod-us-central-0.grafana.net/collect/0b97a9e7c438f4bad009d2afe6d7f754',
  app: {
    name: 'faro-demo-react',
    version: '1.0.0',
    environment: 'production'
  },
});
```

```
// send a log message
import { faro } from '@grafana/faro-web-sdk';

// will be captured
console.info('hello world');

// push log explicitly
faro.api.pushLog(['hello world']);

// will be captured
throw new Error('oh no');

// push error manually
faro.api.pushError(new Error('oh no'));

// push a RUM event
faro.api.pushEvent('click_sign_up_button');
```

▼ Frontend Application에 Faro 설정하기 - 기본

Frontend Application에 Faro 설정하기 - 기본

- 공식 문서 → Faro가 기본적으로 제공하는 데이터보다 더 많은 데이터들을 자유롭게 추가할 수 있습니다.
- 모든 코드는 예시 레포지토리에서 확인할 수 있습니다.
- 여기서는 React 18과 함께 아래의 라이브러리들을 사용합니다.

```
@grafana/faro-core
@grafana/faro-react
@grafana/faro-web-sdk
@grafana/faro-web-tracing
```

Faro 초기화 코드

- Faro를 사용하기 위해서는 우선 코드단에서 Faro를 초기화하고, 관련 설정값들을 지정해야 합니다.
React의 경우, 아래처럼 지정합니다.

```
import {
  createRoutesFromChildren,
  matchRoutes,
  Routes,
  useLocation,
  useNavigationType,
} from "react-router-dom";
import type { Faro } from "@grafana/faro-react";
import {
  initializeFaro as coreInit,
  getWebInstrumentations,
  ReactIntegration,
  ReactRouterVersion,
} from "@grafana/faro-react";
import { TracingInstrumentation } from "@grafana/faro-web-tracing";
import { FARO_URL } from "../settings";

export const initializeFaro = (): Faro => {
  const faro = coreInit({
    url: FARO_URL,
    instrumentations: [
      new TracingInstrumentation(),
      ...getWebInstrumentations({
        captureConsole: true,
      }),
      new TracingInstrumentation(),
      new ReactIntegration({
        router: {
          version: ReactRouterVersion.V6,
          dependencies: {
            createRoutesFromChildren,
```

```

        matchRoutes,
        Routes,
        useLocation,
        useNavigationType,
      },
    },
  )),
],
session: (window as any).__PRELOADED_STATE?.faro?.session,
app: {
  name: "faro-demo-react",
  version: "1.0.0",
  environment: "production",
},
});

faro.api.pushLog(["Faro for faro-demo-react has been
initialized."]);
return faro;
};

```

- 위에서 사용한 `FARO_URL`은 Grafana Cloud에서 생성한 Faro Frontend Application에게 정보를 전달하기 위한 URL입니다.

FaroErrorBoundary 에 대해

- `@grafana/faro-react`에서는 `FaroErrorBoundary`라는 컴포넌트를 제공합니다. 이 컴포넌트는 별다른 작업은 하지 않고, 단지 발생한 에러를 React에 맞게 깔끔하게 변환해주는 작업만을 수행합니다. 아래처럼 `render()`에 넣어줍니다.
- 또한 애플리케이션의 최상단에 위에서 생성한 `initializeFaro()` 함수를 호출해 Faro 관련 초기화를 수행합니다.

```

import React from "react";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
import { FaroErrorBoundary } from "@grafana/faro-react";
import { initializeFaro } from "../global/initializeFaro";
import { createRoot } from "react-dom/client";

initializeFaro();

const container = document.getElementById("root");
const root = createRoot(container!);
root.render(
  <React.StrictMode>
    <FaroErrorBoundary>
      <App />
    </FaroErrorBoundary>
  </React.StrictMode>
);

reportWebVitals();

```

예제 코드 실행 및 결과

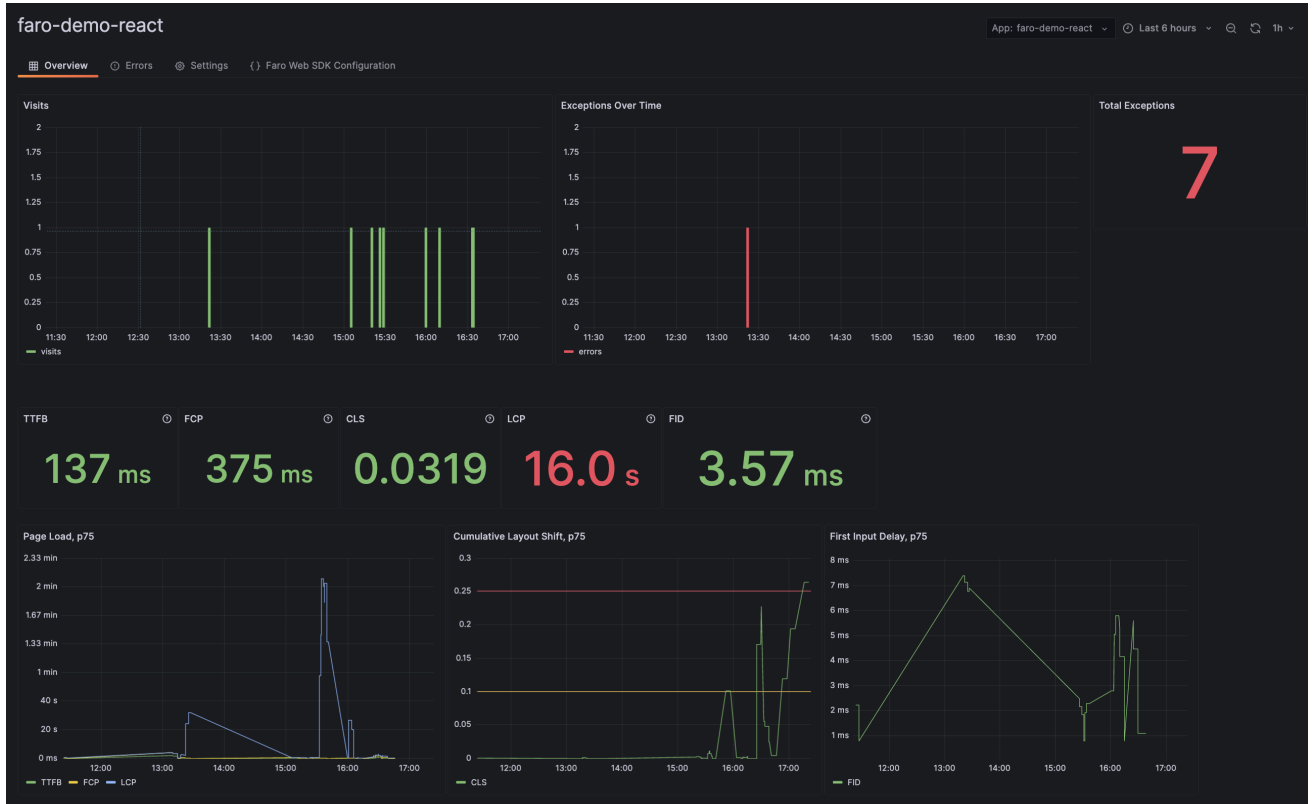
- 예제 코드는 `yarn start` 로 실행해도 되지만, 배포 환경을 흉내내기 위해 아래처럼 실행할 것을 권장합니다.

```

# 1. Docker build
docker build -t faro-demo-react:0.0.1 .
# 2. Docker run
docker run -dp 8000:80 faro-demo-react:0.0.1

```

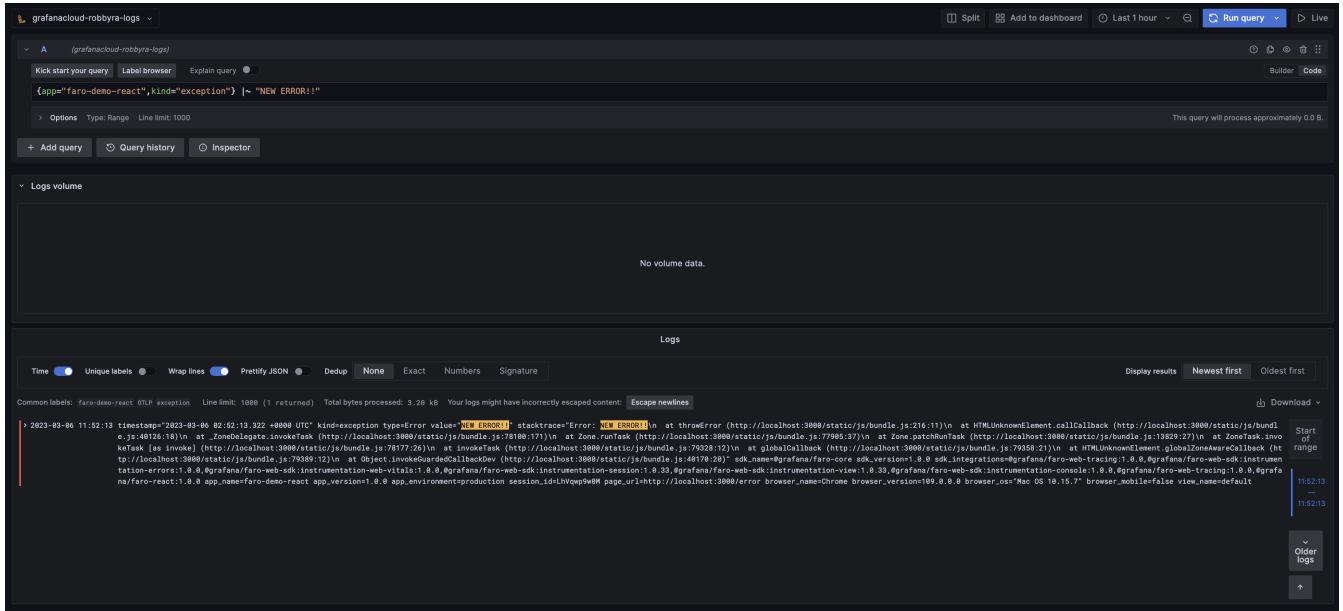
- 위 명령대로 수행하면 `localhost:8000` 에서 아래와 같은 예시 애플리케이션을 확인할 수 있습니다. 실행된 애플리케이션에서 다양한 버튼을 클릭하고, 새로고침하는 등 이벤트를 발생시키면 Faro의 화면 중 Overview 탭에서 아래와 같이 데이터를 확인할 수 있습니다.



- localhost:8080 에서 localhost:8080/error 로 접속하면 throw new Error("NEW ERROR!!") 로 인해 에러가 발생하는데, 따로 catch하는 부분이 없어 콘솔에 에러가 출력됩니다. 이러한 에러들은 Faro 대시보드 중 Errors 탭에서 확인할 수 있습니다.



- 에러 로그는 Loki에 저장되는데, 우측에 NEW ERROR!! 에러를 클릭하면 자동으로 Loki 대시보드로 이동합니다. 이동 후에는 Loki에서 해당 에러가 발생시킨 로그를 확인할 수 있습니다.



▼ Frontend Application에 Faro 설정하기 - Distributed Tracing

Frontend Application에 Faro 설정하기 - Distributed Tracing

- 📌 Distributed tracing의 구현을 테스트하는 용도로 이전에 <https://pyengine.atlassian.net/wiki/spaces/SD/pages/2471133189/1.#4.-%EC%BD%94%EB%93%9C%EC%97%90-tracing-%EC%84%A4%EC%A0%95%ED%95%98%EA%B8%B0> 에서 만든 3개의 서버를 사용했습니다.

라이브러리

- 서버들은 OpenTelemetry를 사용해 distributed tracing을 구현하고 있으며, Grafana Faro 또한 OpenTelemetry와 연계가 원활히 가능합니다. 따라서 아래의 라이브러리들을 설치해 사용합니다.

```
@opentelemetry/api
@opentelemetry/exporter-trace-otlp-http
@opentelemetry/resources
@opentelemetry/sdk-trace-base
@opentelemetry/sdk-trace-web
@opentelemetry/semantic-conventions
```

tracer 설정

- Python에 tracer를 설정하던 것과 마찬가지로, Frontend application에도 tracer 관련 설정을 아래와 같이 해줍니다.

```

import { Resource } from "@opentelemetry/resources";
import { SemanticResourceAttributes } from "@opentelemetry/semantic-
conventions";
import { WebTracerProvider } from "@opentelemetry/sdk-trace-web";
import { BatchSpanProcessor } from "@opentelemetry/sdk-trace-base";
import { OTLPTraceExporter } from "@opentelemetry/exporter-trace-
otlp-http";
import { OTEL_COLLECTOR_URL } from "../settings";

const resource = Resource.default().merge(
  new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: "faro-demo-react",
    [SemanticResourceAttributes.SERVICE_VERSION]: "0.0.0",
  })
);

const provider = new WebTracerProvider({
  resource: resource,
});

const exporter = new OTLPTraceExporter({ url: OTEL_COLLECTOR_URL });
const processor = new BatchSpanProcessor(exporter);

provider.addSpanProcessor(processor);

provider.register();

```

- 여기서 생성한 tracer는 Faro가 frontend application 자체에 대한 trace만을 위해 사용됩니다.

Tracing 설정

- Distributed tracing을 구현하기 위해서는 서로 다른 서비스들이 같은 상위 span 내에 있음을 나타낼 수 있어야 합니다. HTTP 기반의 소통 방식의 경우, `traceparent` 라는 header를 전달함으로써 이를 나타냅니다.
- Frontend application이 처음으로 요청을 발생시키는 주체가 되기 때문에, 이 부분에서 `traceparent` 의 값을 지정해 request header로 전달해야 합니다.
- Axios를 사용하는 경우, 아래와 같이 설정할 수 있습니다.


```

import axios, { AxiosRequestConfig } from "axios";
import { ITEM_API } from "../settings";
import { WeatherResponse, ItemResponse } from "../interface";
import { context, propagation } from "@opentelemetry/api";

// validateItem `ITEM_API`
// traceparent header span
export const validateItem = async (
  name: string,
  price: number
): Promise<ItemResponse> => {
  const carrier: TraceCarrier = { traceparent: "" };
  propagation.inject(context.active(), carrier);
  const Axios = axios.create();
  const response = await Axios.post<ItemResponse>(
    `${ITEM_API}`,
    {
      name,
      price,
    },
    {
      headers: {
        traceparent: carrier.traceparent,
      },
    }
  );
  console.log(`response.data: ${JSON.stringify(response.data)}`);
  return response.data;
};

```

