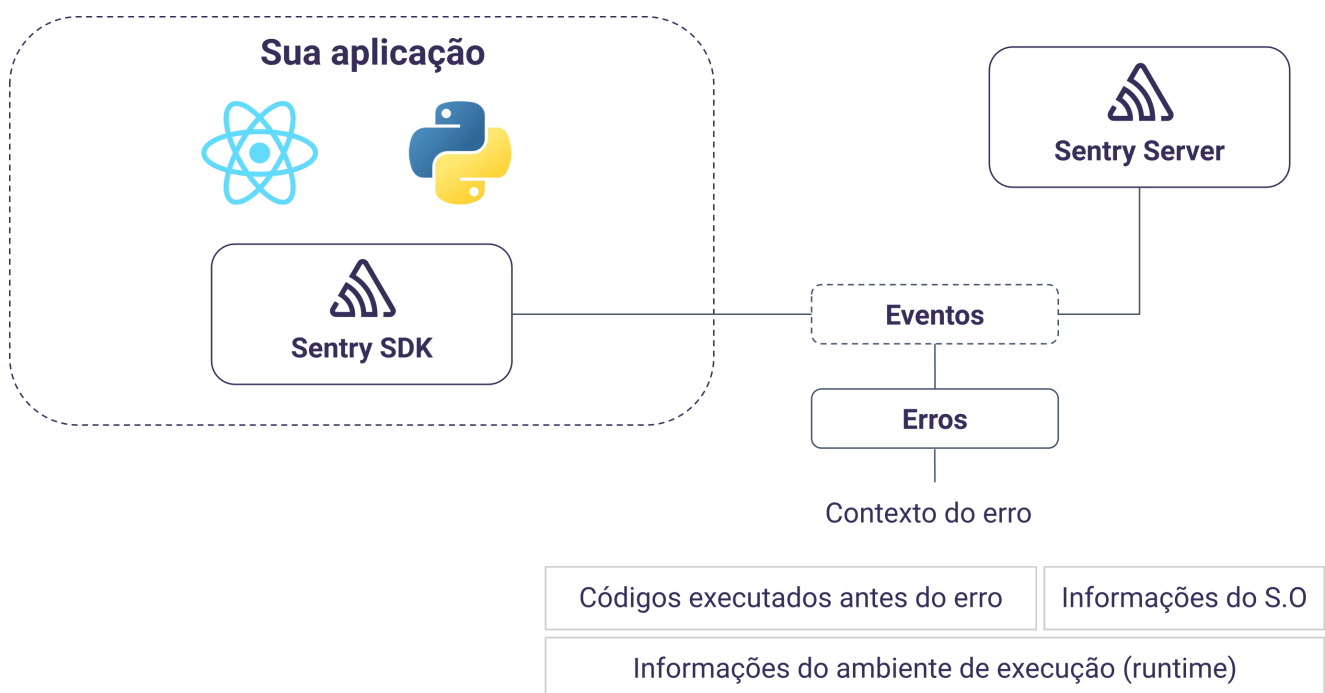


Introdução prática ao Sentry

Este documento faz a apresentação prática dos principais conceitos relacionados ao Sentry e sua utilização. Para isso, primeiro faz-se uma introdução geral ao funcionamento do Sentry. Em seguida, são apresentados exemplos de uso do Sentry nas linguagens de programação Python e Javascript.

Fluxo de funcionamento

Antes de iniciar o conteúdo deste material, é importante entender a forma de funcionamento do Sentry e os conceitos que a ferramenta utiliza nesse processo. Na figura abaixo, tem-se o fluxo geral de funcionamento do Sentry.



Como pode-se observar, existem duas peças principais em um cenário de funcionamento do Sentry:

- Sentry Server: Servidor Sentry que é o responsável por armazenar erros recebidos dos clientes e os armazenar de modo que seja fácil para os desenvolvedores realizarem sua avaliação;
- Sentry SDK: O Sentry SDK representa o cliente do Servidor Sentry. Esse componente, coleta os eventos da aplicação, suas informações e contexto onde ocorreram, e os envia para o Sentry Server. Existem atualmente, mais de **100 SDKs disponíveis no Sentry** [🔗](#). Todos esses são personalizados para coletar informações específicas de cada ambiente a qual se integram. No entanto, deve-

se notar que os desenvolvedores, na medida do possível, buscam manter a API dos SDKs parecidas, de modo que a curva de aprendizado não seja muito grande. Ao aprender a utilizar um SDK Sentry, você provavelmente aprendeu a utilizar vários outros.

Nos tópicos abaixo, faremos a utilização prática do Sentry! Vamos lá!

Sentry SDK - Python

Esta seção, apresenta exemplos de utilização do cliente Sentry para a linguagem de programação Python. Para isso, primeiro faz-se a configuração do ambiente Python e a instalação do cliente. Em seguida, é realizado um exemplo básico de utilização do cliente. Por fim, um exemplo de utilização do cliente Sentry integrado ao microframework Flask é fornecido.

Ambiente

“ A construção dessa documentação foi realizada utilizando Python 3.8 no sistema operacional Ubuntu 20.04. Mudanças podem ser necessárias a depender de seu ambiente.

Para realizar a criação do ambiente de desenvolvimento Python que será utilizado, primeiro faça a criação de um ambiente virtual:

```
python3 -m venv venv
```

sh

Agora, ative o ambiente criado e faça a atualização dos pacotes `pip`, `wheel` e `setuptools`:

```
pip3 install --upgrade pip wheel setuptools
```

sh

Com seu ambiente atualizado, instale o Sentry SDK:

```
pip3 install sentry-sdk
```

sh

Aproveite e faça a instalação do Flask, que será utilizado mais tarde neste tutorial:

```
pip3 install Flask
```

py

Pronto! Agora, você pode começar o tutorial.

Exemplo básico

Agora, podemos começar o exemplo básico de utilização do Sentry SDK, instalado no passo anterior. Então, comece fazendo a criação de um arquivo Python de nome `app.py`. Neste arquivo, coloque o seguinte conteúdo:

```
for i in range(750, 790):  
    print(i, end = ' | ')  
  
    if i == 777:  
        raise ValueError("Número não permitido!")
```

py

Ao executar esse [script](#), como esperado, será gerado um erro. Até aqui, nada de novo para nós. No entanto, queremos que esse erro seja enviado para o Sentry, de modo que possamos fazer o uso de todas as funcionalidades da ferramenta.

Para que isso seja possível, primeiro precisamos configurar o Sentry SDK em nossa aplicação. Faça isso através da importação do SDK e em seguida o inicialize:

```
# 1. Importando o SDK  
import sentry_sdk  
  
# 2. Inicializando o SDK  
sentry_sdk.init(  
    dsn = "<SEU-DSN-MÁGICO-AQUI>"  
)  
  
# 3. Código Python com erro  
for i in range(750, 790):  
    print(i, end = ' | ')  
  
    if i == 777:  
        raise ValueError("Número não permitido!")
```

py

“ Lembre-se, o **DSN** requerido no código acima deve ser gerado em seu servidor Sentry.

Agora, com o código atualizado, faça a execução novamente. Você perceberá que o erro ainda estará lá, mas agora, uma nova mensagem surgiu ao final da execução de seu código. Algo como apresentado abaixo:

```
Sentry is attempting to send 2 pending error messages
Waiting up to 2 seconds
Press Ctrl-C to quit
```

sh

Essa mensagem indica que o Sentry SDK está trabalhando! O que ele fez foi coletar o evento de erro gerado e o enviou para o Sentry Server. De uma olhada lá!

“ Agora, uma questão importante: Como isso foi feito ?

Essa é uma das mágicas que os desenvolvedores do Sentry fazem. Eles abstraem toda a complexidade de coleta e envio de eventos de erro para o servidor. Para isso, o SDK se integra com a API da linguagem e faz com que todas as execuções não tratadas sejam enviadas para o Sentry. Nesse envio, informações relevantes do ambiente são consideradas, como versões das bibliotecas instaladas no ambiente, usuários autenticados (aplicações [web](#)) entre outras.

Enriquecendo eventos

Os desenvolvedores do Sentry fazem o possível para que as informações que eles coletam no evento sejam o suficiente para boa parte das aplicações. No entanto, pode ser que contextos específicos precisem de informações que não estão disponíveis nos eventos padrão, nesse caso, podemos fazer o [Enriquecimento dos eventos](#).

Com esse [enriquecimento](#), podemos adicionar informações customizadas aos eventos. Existem muitas formas de fazer isso, dentre elas:

1. [Adição de contexto](#);
2. [Identificação dos usuários](#);
3. [Tags customizadas](#);
4. [Breadcrumbs](#).

As subseções abaixo, apresentam exemplos de uso de cada uma dessas formas de enriquecer as informações dos eventos.

Adição de contexto

A adição de contexto, permite que você faça a adição de dados arbitrários ao evento, o que inclui dicionários de dados, arquivos binários e objetos de exceção.

“ Embora seja possível fazer o envio de arquivos binários, a [documentação do Python Sentry SDK](#) [indica](#) que essa opção deve ser evitada. Limites de tamanho para os eventos são aplicados, e caso excedidos, o evento não é registrado no Sentry.

Para fazer a adição de contexto, você precisa utilizar a função `set_context`, disponível no objeto `sentry_sdk`. Essa função recebe o nome do contexto e seu conteúdo.

“ O nome do contexto não possui regras, sendo aceito qualquer valor do tipo `string`. Por outro lado, o conteúdo do contexto deve sempre ser representado como um dicionário.

O código abaixo exemplifica a criação de um contexto de nome `maquina`, que possui um objeto `dicionário` simples com seu conteúdo:

```
# 1. Importando o SDK
import sentry_sdk

# 2. Inicializando o SDK
sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>"
)

# 3. Configurando contexto
sentry_sdk.set_context("maquina", dict(
    nome = "Minha máquina legal",
    cpus = 16,
    ram = 24
))
```

py

Agora, ao fazer o uso do `script`, você verá que no evento salvo no Sentry, tem-se um novo contexto `maquina` adicionado.

Caso você necessite adicionar uma lista de valores ao contexto, seguindo a regra mencionada anteriormente, essa lista deverá estar dentro de um dicionário. O exemplo

abaixo apresenta a criação de um contexto de nome `arquivos` com uma lista de valores associado:

```
sentry_sdk.set_context("arquivos", dict(
    instances = [
        {
            "id": 1,
            "name": "file_1.txt"
        },
        {
            "id": 2,
            "name": "file_2.txt"
        }
    ]
))
```

py

Identificação de usuário

Uma informação que pode ser útil no estudo do erro é o usuário que está utilizando a aplicação. Com essa informação, pode-se entender quais e quantos são os usuários afetados por um erro.

No Sentry SDK é possível fazer a identificação do usuário que está utilizando o sistema. Para a identificação do usuário que está logado, é possível fazer o uso da função `set_user` disponível no objeto `sentry_sdk`. Essa função deve receber um `dicionário` com no mínimo, uma das seguintes chaves:

- `id` ⓘ: Identificador interno do usuário no sistema
- `username` ⓘ: Nome do usuário
- `email` ⓘ: Email do usuário;
- `ip_address` ⓘ: IP do usuário.

Para testar, vamos fazer a adição de um usuário com os campos `id` e `username`:

```
# 1. Importando o SDK
import sentry_sdk

# 2. Inicializando o SDK
sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>"
)

# 3. Definindo o usuário
```

```
sentry_sdk.set_user(dict(
    id = 1,
    username = "gsansigolo"
))
```

py

Com isso, todos os eventos gerados em nossa aplicação de exemplo, terão o usuário `gsansigolo`.

“ A identificação dos usuários e o armazenamento de suas informações no Sentry deve estar de acordo com a Lei Geral de Proteção de Dados Pessoais (LGPD). Para esses casos, o Sentry SDK oferece opções que auxiliam na identificação e remoção de informações sensíveis dos eventos antes de seu envio para o Sentry Server. Mais informações podem ser encontradas na [documentação oficial](#) 📄

Tags customizadas

Quando os eventos são enviados para o Sentry Server, eles são indexados em um sistema de busca, para que análises possam ser feitas. Com base nessa funcionalidade, a fim de auxiliar os desenvolvedores na criação de *insights*, o Sentry permite a associação de *tags* aos eventos gerados. Nessas *tags*, pode-se definir informações arbitrárias que facilitem a identificação do erro e seu contexto. Por padrão, o Sentry faz a criação das seguintes *tags*:

- *environment* 📄: *Tag* padrão do Sentry para identificar o ambiente onde o código está sendo executado (e.g., *Produção*, *Homologação* e *Desenvolvimento*). O valor definido nessa *tag* é arbitrário;
- *release* 📄: *Tag* padrão do Sentry para identificar a versão do código.

Por essas serem *tags* padrão, elas são geradas não apenas no SDK Python, mas também em todos os SDKs disponíveis para uso do Sentry. No entanto, deve-se considerar que, os SDKs podem gerar *tags* específicas para o ambiente/projeto/linguagem de programação que está sendo utilizado. Por exemplo, no caso do SDK do Python, tem-se as seguintes *tags* padrão associadas ao evento:

- *handled*: Indica se a exceção foi tratada ou não no código que a gerou;
- *runtime*: Ambiente de execução (e.g., *CPython 3.9.7*);
- *runtime.name*: Nome do ambiente de execução (e.g., *CPython*);

- `server_name`: Nome da máquina onde o erro ocorreu (e.g., `Linux`).

“ Na seção anterior, fez-se a definição do usuário. Para facilitar o uso dessa informação, o SDK Python define o `id` do usuário como uma `tag`.

Além das `tags` padrão, valores customizados podem ser definidos. Para isso, você pode utilizar a função `set_tag` disponível no objeto `sentry_sdk`. O código abaixo, apresenta um exemplo da definição da `tag` de nome `cor` com o valor `vermelho`:

```
# 1. Importando o SDK
import sentry_sdk

# 2. Inicializando o SDK
sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>"
)

# 3. Definindo a tag customizada
sentry_sdk.set_tag("cor", "vermelho")
```

py

Breadcrumbs

Uma das formas de entender o que causou o erro é a identificação dos passos que foram tomados pelo usuário (e pelo código) até chegar ao erro. Como forma de disponibilizar essa informação nos eventos gerados, o Sentry suporta os chamados `breadcrumbs`. Esses, representam a trilha de eventos que antecederam o erro.

Uma vez definidas, as `breadcrumbs` são apresentadas visualizadamente em uma linha do tempo, de modo a formar a representação dos passos seguidos até a ocorrência do erro.

“ Os `breadcrumbs`, são definidos pelos desenvolvedores do Sentry, como o equivalente ao "log tradicional" no Sentry.

Com isso, a forma de criação de uma `breadcrumb` em um evento, deve ser pensada e feita da mesma forma que no sistemas de log tradicional: Para cada linha de código executada antes do erro, deve-se definir explicitamente o que ela representa e seu tipo/categoria.

Para ver como isso funciona na prática, primeiro, crie um arquivo de nome `visual.py` com o seguinte conteúdo:

```
# 1. Importando o SDK
import sentry_sdk

# 2. Inicializando o SDK
sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>"
)

# 3. Definindo o bloco de código que será executado
print('Passo 1')

print('Passo 2')

print('Passo 3')

print('Passo 4')

print(' | '.join([1, 2, 3])) # TypeError (sorry)
```

py

Como pode-se perceber no código definido acima, o último print a ser executado irá gerar uma exceção (`TypeError`). Nesse caso, há várias linhas de código que antecedem o evento de erro. Vamos então, criar uma `breadcrumbs` que mostre cada uma dessas linhas executadas até a geração do erro.

Para criar um "rastro" ou registro na `breadcrumbs`, você pode utilizar a função `add_breadcrumb`, disponível no objeto `sentry_sdk`:

```
# 1. Importando o SDK
import sentry_sdk

# 2. Inicializando o SDK
sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>"
)

# 3. Definindo o bloco de código que será executado
sentry_sdk.add_breadcrumb(
    category='console.stdout',
    message = 'Executando passo 1',
    level = 'info'
)

print('Passo 1')
```

```

sentry_sdk.add_breadcrumb(
    category = 'console.stdout',
    message = 'Executando passo 2',
    level = 'info'
)
print('Passo 2')

sentry_sdk.add_breadcrumb(
    category = 'console.stdout',
    message = 'Executando passo 3',
    level = 'info'
)
print('Passo 3')

sentry_sdk.add_breadcrumb(
    category = 'console.stdout',
    message = 'Executando passo 4',
    level = 'info'
)
print('Passo 4')

print(' | '.join([1, 2, 3])) # TypeError (sorry)

```

py

Agora, execute o código e veja o evento que foi gerado em seu Sentry Server. Como é possível perceber, agora, você tem uma nova seção na sua interface de inspeção de eventos nomeada **Breadcrumbs**.

No código acima, vale ressaltar que o argumento **category** pode receber qualquer *string*, não tendo valores pré-definidos. Ao contrário disso, o campo **level** precisa seguir a lista de **valores padrão** [↗](#).

“ Como você pode notar, o uso e definição das breadcrumbs é similar ao que é feito no módulo **logging** [↗](#), por exemplo. Para cada informação que se deseja armazenar, uma execução da função **add_breadcrumb** deve ser realizada.

Para uso dos **breadcrumbs**, o Sentry SDK oferece além das opções apresentadas, outras configurações. Por exemplo, pode-se alterar nível de severidade e até mesmo definir de forma arbitrária o timestamp em que o registro . Para obter mais informações, por favor, consulte a seção **Using Breadcrumbs** [↗](#) da documentação.

Integrations

Até aqui, as operações que realizamos foram bem **manuais**. Isso ocorre já que estávamos utilizando as funcionalidades base oferecidas pelo Sentry SDK para criar e manipular os eventos de erro.



No entanto, há uma notícia positiva: A menos que você precise dos comandos mostrados anteriormente para casos específicos, eles não precisam ser utilizados em seu dia a dia para que o Sentry seja integrado em sua aplicação. Ao contrário disso, o padrão "apenas importe e uso" é um grande objetivo dos desenvolvedores Sentry.

Para que isso seja possível, o Sentry SDK oferece **Integrations** [🔗](#). Esses componentes, quando definidos no código, utilizam a API base do Sentry SDK, que vimos anteriormente, e automaticamente configura **breadcrumbs**, **identificação de usuários** e **contextos especializados para diferentes tipos de ambientes computacionais**. Além disso, as **integrations** também permitem a configuração automatizada de bibliotecas de terceiros em nosso código. Por exemplo, para utilizar o Sentry junto ao Flask, não é preciso fazer definição de usuário nem de contexto, a integração fornecida no SDK trata desses detalhes automaticamente.



“ Quer uma curiosidade ? No começo desse documento, vimos que quando uma exceção é lançada, ela é automaticamente colocada em um evento e enviada ao Sentry. Até parecia mágica, mas na verdade são **Integrations**. Nos bastidos do Sentry SDK, esse comportamento só ocorre já que há uma Integration pré-definida que configura como tratar as exceções. Para mais detalhes, consulte a **documentação oficial** [🔗](#).

No Sentry SDK Python, existem muitas integrações que podemos utilizar, sendo algumas delas:

- **Apache Airflow** [🔗](#): Integração com Apache AirFlow. Nessa integração, são definidas **tags**, contextos e **breadcrumbs** exclusivos ao Airflow;
- **Celery** [🔗](#): Integração com o Celery. Auxilia os desenvolvedores no controle de contextos de eventos que ocorrem nas execuções distribuídas;
- **Flask** [🔗](#): Integração com o Flask. Facilita o uso do ecossistema Flask e sua integração com o Sentry. Estão disponíveis, por exemplo, opções para identificação de usuários via **Flask-Login** [🔗](#) e **breadcrumbs** SQL via **SQLAlchemy** [🔗](#).

- **Logging** : Integração com o módulo **logging do Python**  para a criação de eventos de erro.

Exemplo: Logging Integration

Para exemplificar a forma de uso das integrações, vamos fazer o uso da integração **Logging** . Essa integração, conforme mencionado anteriormente, utiliza as informações geradas com o módulo **logging do Python**  para a criação de **breadcrumbs**.

Para começar, primeiro crie um arquivo de nome **integration.py**. Nesse arquivo, vamos fazer a configuração da integração. Para isso, primeiro importe o módulo **logging** e a integração **Logging**:

```
import logging
from sentry_sdk.integrations.logging import LoggingIntegration
```

py

Agora, utilizando o objeto da integração (**LoggingIntegration**), defina as configurações do **logging** que serão utilizadas:

```
loginfo = LoggingIntegration(
    event_level=logging.ERROR # envia os erros do logging como eventos do
    Sentry
)
```

py

Agora, utilizando o objeto criado, defina na função de inicialização do Sentry(**init**), a integração realizada:

```
import sentry_sdk
import logging
from sentry_sdk.integrations.logging import LoggingIntegration

loginfo = LoggingIntegration(
    event_level=logging.ERROR # envia os erros do logging como eventos do
    Sentry
)

sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>",
    integrations=[loginfo] # lista de integrações =)
)
```

py

Agora, para testar se tudo está funcionando, vamos adicionar um código em que fazemos a criação de eventos de erros através do **logging**:

```
import sentry_sdk
import logging
from sentry_sdk.integrations.logging import LoggingIntegration

loginfo = LoggingIntegration(
    event_level=logging.ERROR # envia os erros do logging como eventos do
    Sentry
)

sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>",
    integrations=[loginfo] # lista de integrações =)
)

# Criando eventos de erro
logging.error("Errou!")
logging.error("Errou! (x2)")
```

py

Para adicionar informações extras ao evento criado, você pode utilizar o parâmetro **extra**:

```
logging.error("Errou com extra args!", extra = dict(value = 777))
```

py

Dê uma olhada em seu Sentry Server! Agora, as informações de evento e **breadcrumb** estão sendo registradas através do módulo logging. Essa integração é muito útil quando sua aplicação já utiliza esse módulo e você deseja apenas redirecionar as informações.

“ Lembre-se de que, o Sentry não foi feito para armazenar Logs. Então, estude seu cenário antes de apenas "mudar tudo".

Exemplo com Flask

Agora que já fizemos um tour pelas principais funcionalidades e conceitos envolvidos na utilização do Sentry e Sentry SDK, podemos começar a avançar nas aplicações.

Para isso, nesta subseção, veremos como o Sentry pode ser utilizado em aplicações Flask através da **integration** Flask disponível no Sentry Python SDK.

Para esse exemplo, iremos criar uma API Rest simples, que recebe um documento JSON e o devolve com um campo extra. Comece criando um arquivo de nome **api.py** com o seguinte conteúdo:

```
from flask import Flask, request, jsonify

app = Flask(__name__)
app.config['JSON_AS_ASCII'] = False # UTF-8

@app.route('/', methods = ['POST'])
def apijson():

    # 1. Recuperando dados
    data = request.json

    # 2. Adicionando campo extra
    data['extra'] = 'Há uma missão secreta no espaço não mapeado. Vamos lá'

    return jsonify(data)

if __name__ == '__main__':
    app.run()
```

py

Após executar o **script** **api.py**, faça uma requisição de teste e verifique se está tudo funcionando corretamente:

```
curl -X POST http://127.0.0.1:5000 \
  -H 'Content-Type: application/json' \
  -d '{"message": "hello ?"}'
```

py

A resposta deve ser algo como:

```
{
  "extra": "Há uma missão secreta no espaço não mapeado. Vamos lá",
  "message": "hello ?"
}
```

sh

Agora, utilizando a **integration** Flask do Sentry Python SDK, vamos adicionar o Sentry ao Flask. Para isso, adicione ao seu arquivo **api.py** o seguinte trecho de código:

```
from flask import Flask, request, jsonify

import sentry_sdk
from sentry_sdk.integrations.flask import FlaskIntegration

sentry_sdk.init(
    dsn = "<SEU-DSN-MÁGICO-AQUI>",
    integrations=[FlaskIntegration()]
)

app = Flask(__name__)
# omitido ...
```

py

“ Lembre-se de que, seguindo as boas práticas indicadas na documentação, você deve criar um **Projeto** [🔗](#) diferente para essa aplicação no Sentry. Não utilize o mesmo **DSN** dos exemplos anteriores.

Pronto! Agora, sua aplicação já está integrada ao Sentry. Para fazermos um teste dos eventos de erro, vamos modificar o código da função **apijson**. Neste código, vamos gerar um erro:

```
@app.route('/', methods = ['POST'])
def apijson():

    # 1. Recuperando dados
    data = request.json() # erro aqui

    # 2. Adicionando campo extra
    data['extra'] = 'Há uma missão secreta no espaço não mapeado. Vamos lá'

    return jsonify(data)
```

py

Agora, tente enviar o dado novamente para a API. Você receberá o seguinte erro:

```
<!doctype html>
<html lang=en>
<title>500 Internal Server Error</title>
```




```
<h1>Internal Server Error</h1>
```

```
<p>The server encountered an internal error and was unable to complete  
your request. Either the server is overloaded or there is an error in the  
application.</p>
```



html


Consulte o Sentry Server. Possivelmente você verá um novo evento de erro. Nesse evento, note que, informações relacionadas ao método HTTP utilizado, URL, headers e os dados recebidos estão todos no evento. Isso é feito automaticamente pela [integration](#) do Flask fornecida pelo Sentry. Essa integração facilita muito a utilização do Sentry em aplicações que já existem, evitando modificações e adequações nos códigos.

Caso você deseje, foram preparados outros exemplos de utilização do Flask com o Sentry. Eles estão disponíveis nos seguintes repositórios:



1. [Flask Basic Example](#) : Exemplo simples de integração entre o Flask e o Sentry;
2. [My Users API](#) : Exemplo de integração entre o Flask e o Sentry utilizando [Application Factories](#) .


Sentry SDK - React

O Sentry, conforme mencionado no início dessa documentação, fornece diversos SDKs que se integram as linguagens de programação/ferramentas/ambientes para capturar eventos de erros e os enviar para o Sentry Server. Dentre esses SDKs, existem versões específicas para tecnologias front-end, como [Angular](#)  e [React](#) .

Nesta seção, será feita uma rápida introdução a esses SDKs front-end, com foco no uso do React. Além disso, o recurso de Sentry [User Feedback](#) , muito útil para aplicações front-end, será apresentado.

Ambiente e template do projeto

Para acompanhar os passos que serão apresentados, você deverá ter o [Node.js](#) , com versão 16.0 ou superior. Para o gerenciamento de pacotes, será feito uso do [Yarn](#) .

Com as dependências base instaladas em sua máquina, utilizando o [vite.js](#) , faça a geração de um novo projeto React:

```
yarn create vite
```

sh

Nos campos a serem preenchidos, utilize as seguintes informações:

Project name: example-sentry-app

Select a framework: react

Select a variant: react

Pronto! Agora, com o diretório `example-sentry-app` criado, acesse-o. No diretório, instale as dependências com o comando `yarn`:

```
yarn
```

sh

Aproveite e instale o Sentry React SDK:

```
yarn add @sentry/react @sentry/tracing
```

sh

Em seguida, delete todos os arquivos do diretório `src`. Por fim, criei um novo arquivo chamado `src/main.jsx` com o seguinte conteúdo:

```
import React from "react";
import ReactDOM from "react-dom/client";

const MyComponent = () => (
  <>
    <button
      onClick={() => {
        alert("Hello");
      }}
    >
      Clique aqui
    </button>
  </>
);

ReactDOM.createRoot(document.getElementById("root")).render(<MyComponent
/>);
```

js

Após a configuração, execute o projeto:

```
yarn dev
```

sh

A resposta desse comando deverá ser algo como:

```
vite v2.9.9 dev server running at:  
  
> Local: http://localhost:3000/  
> Network: use `--host` to expose  
  
ready in 302ms.
```

sh

Acesse o endereço `http://localhost:3000/` em seu navegador. Você deverá ver um botão na página. Ao clicar nele, um `alert` será apresentado.

Fazendo isso, você está pronto para integrar o Sentry a sua aplicação React.

Integrando Sentry ao React

Para fazer a integração entre o Sentry e o React, de forma análoga ao processo realizado no Python, tudo o que precisamos fazer é importar o SDK e em seguida fazer sua inicialização. Para isso, em seu arquivo `src/main.jsx` adicione o seguinte bloco de código:

```
import React from "react";  
import ReactDOM from "react-dom/client";  
  
import * as Sentry from "@sentry/react";  
  
Sentry.init({  
  dsn: "<SEU-DSN-MÁGICO-AQUI>",  
});  
  
// Omitido...
```

js

Notou algo de similar nesse código ? Além dos passos lógicos serem os mesmos que os apresentados no Python, a forma de utilização também é a mesma: Importe o SDK, faça sua inicialização e se necessário, defina `integrations`. Isso diminui a curva de aprendizado do Sentry, tornando-o simples de ser integrado.


Bem, para verificar se está tudo funcionando, vamos introduzir um erro em nosso código. Para isso, na função callback do botão dentro de `MyComponent`, adicione o seguinte código:

```
const MyComponent = () => (
  <>
    <button
      onClick={() => {
        funçãoQueNãoExiste(); // erro!
      }}
    >
      Clique aqui
    </button>
  </>
);
```


js

Agora, execute o código, clique no botão e consulte seu Sentry Server. Você verá que o erro foi gerado com muitas informações sobre o contexto, incluindo o **breadcrumbs** da interface com o botão que causou o erro.

User Feedback

Para finalizar essa documentação, nesse último exemplo, será mostrado como a funcionalidade de **User Feedback**  pode ser adicionada em uma aplicação React.

Com o uso do User Feedback, o Sentry React SDK cria um formulário de erro na tela do usuário. Nesse formulário o usuário pode auxiliar os desenvolvedores passando mais informações sobre o que estava sendo feito na aplicação no momento em que o problema ocorreu.

Para utilizar essa funcionalidade, tudo o que precisamos fazer é definir o **hook** **beforeSend**  na inicialização do Sentry. Esse método é invocado toda vez antes de um evento ser enviado para o Sentry Server. Dentro deste método, faremos a definição de um formulário, no qual coletaremos a descrição do usuário. Em seguida, as informações coletadas são enviadas para o Sentry Server e vinculadas ao evento de erro.


Seguindo a ideia do parágrafo anterior, a adição dessa funcionalidade pode ser feita com a seguinte modificação na aplicação React:

```
Sentry.init({
  dsn: "<SEU-DSN-MÁGICO-AQUI>",
  beforeSend: (event, hint) => {
    if (event.exception) {
      Sentry.showReportDialog({ eventId: event.event_id });
    }
  }
});
```

```
    return event;  
  },  
});
```

js

Agora, ao gerar um erro, você verá que um widget de formulário será exibido. Esse poderá ser preenchido pelos usuários para que ações possam ser tomadas pelos desenvolvedores.

Caso você deseje, também preparamos um exemplo em que o React é utilizado para consumir uma API. Para saber mais, consulte o repositório [My Users App](#) . Esse app, consome a API feita em Python e Flask na seção anterior.