Olá, amigos e amigas! Hoje trago mais um conteúdo sobre logs — mas, dessa vez, com o nosso querido Flutter.

Ok, você deve estar se perguntando: *qual é a diferença?* Enviar logs não é só fazer uma requisição para a API e pronto?
E, para ser sincero, se você pensa assim… está parcialmente certo.

Porém, existem alguns detalhes importantes aqui. Primeiro: você sabe o que está realmente sendo enviado? Tem ideia de como esses dados estão sendo processados?
Complicou, né?

Pois bem, a ideia deste tutorial vai muito além de apenas enviar logs via HTTP. A proposta é definir um padrão e um formato correto, que seja facilmente interpretado e que não exija manipulações bizarras para ser apresentado ou metrificado (obter métricas).

Então dada a nossa introdução bora colocar a mão na massa.

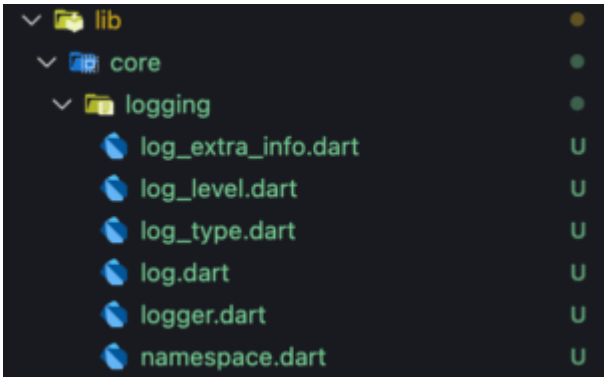
E caso queira também pode acompanhar pelo youtube

Porem antes de darmos inicio precisamos instalar o package Dio para conseguir realizar as requisições http

use o comando: flutter pub add dio

detalhes podem ser obtidos aqui!

Logo de inicio vamos definir a estrutura para nossos logs:



Bem então a primeira coisa a fazer é criar nossas classes de modelo

Vamos criar a classe log.dart

```
import 'dart:convert';
import 'log_level.dart';
import 'log_type.dart';
import 'namespace.dart';

class Log {
  final LogType logType;
  final String acronym;
  final LogLevel level;
  final String serviceName;
  final Namespace namespace;
  final String operation;
  final dynamic content;
  final int? duration;
  final double? value;
  final LogExtraInfo? extra;

  Log({
    required this.logType,
    required this.acronym,
    required this.level,
    required this.serviceName,
    required this.namespace,
    required this.operation,
    required this.content,
    this.duration,
    this.value,
    this.extra,
```

```
  });

  Map<String, dynamic> toJson() {
    return {
      'timestamp': DateTime.now().toUtc().toIso8601String(),
      'logType': logType.name,
      'acronym': acronym,
      'level': level.name,
      'service.name': serviceName,
      'service.namespace': namespace.name,
      'operation': operation,
      'content': content is String ? content : jsonEncode(content),
      if (duration != null) 'duration': duration,
      if (value != null) 'value': value,
      if (extra != null) ...extra!.toJson(),
    };
  }
}
```

Agora vamos criar a classe de suporte log_extra_info.dart

> Importante essa classe ela deve respeitar a sua regra de negocio então tenha em mente que ela deve possuir informações que façam sentido a sua realidade. Outro ponto que gosto de usar em classes de suporte ao log e deixar seus valores opcionais mas pensando sempre em como ela pode apresentar os dados de negocio, troubleshooting e falhas criticas (até o stackTrace completo sem quebrar o log todo).

```
class LogExtraInfo {
  final String? userId;
  final String? transactionId;
  final int? statusCode;
  final dynamic message;
  final String? exception;
  final String? errorMessage;
  final List<String>? errorStack;

  LogExtraInfo({
    this.userId,
    this.transactionId,
    this.statusCode,
```

```
    this.message,
    this.exception,
    this.errorMessage,
    this.errorStack,
  });

  Map<String, dynamic> toJson() {
    return {
      if (userId != null) 'userId': userId,
      if (transactionId != null) 'transactionId': transactionId,
      if (statusCode != null) 'statusCode': statusCode,
      if (message != null) 'message': message,
      if (exception != null) 'exception': exception,
      if (errorMessage != null) 'errorMessage': errorMessage,
      if (errorStack != null) 'errorStack': errorStack,
    };
  }
}
```

E claro os nossos enums: log_level.dart, log_type.dart e namespace.dart

> Apenas para entendimento geral é padrão de mercado assim como boa praticar ter uma segmentação do LogType nestes quatro tipos: Troubleshooting, Audit, Security e Business para uma identificação correta e aplicabilidade do log.

```
enum LogType {
  TBS, // Troubleshooting
  AUD, // Audit
  SEC, // Security
  BUS, // Business
}
```

```
enum LogLevel {
  DEBUG,
  INFO,
  WARN,
  ERROR,
}
```

```
enum Namespace {
  prd,
```

```
  uat,
  des
}
```

Agora um ponto bem importante, no meu exemplo estou carregando as credenciais a partir de um objeto na raiz do projeto, aqui fica a sua vontade/necessidade se quiser carregar de outra forma por exemplo como o dotend, aqui vou deixar um package que uso!

Se quiser seguir a mesma forma

Crie um arquivo chamado secrets.json na raiz do projeto, e adicione o seguinte conteúdo, com suas respectivas API e secret:

```
{
    "URL_LOG_API": "<SUA API PARA INGESTÃO DE LOGS>",
    "URL_LOG_API_DEV": "<SUA KEY PARA INGESTÃO DE LOGS>"
}
```

E por fim vamos criar o arquivo responsável por fazer o envio dos logs o logger.dart

> IMPORTANTE: Observe que a classe necessita de dois valores iniciais sendo a URL e a apiKey, ambas estão definidas no arquivo anterior, mas fique a vontade para alterar caso necessário.

```
import 'package:dio/dio.dart';

class Logger {
  static late Dio _dio;

  static void initialize({required String endpoint, required String apiKey}) {
    _dio = Dio(BaseOptions(
      baseUrl: endpoint,
      headers: {
        'Content-Type': 'application/json; charset=utf-8',
        'Authorization': 'Api-Token $apiKey',
      },
      connectTimeout: const Duration(seconds: 5),
      receiveTimeout: const Duration(seconds: 5),
```

```
    ));
  }

  static Future<void> send(Log log) async {
    try {
      final response = await _dio.post(
        '',
        data: log,
      );

      if (response.statusCode != null && response.statusCode! > 299) {
        print(
            'Erro ao enviar log: ${response.statusCode} -
${response.statusMessage}');
      }
    } on DioException catch (e) {
      print('Erro de Dio ao enviar log: ${e.message}');
    } catch (e) {
      print('Erro desconhecido ao enviar log: $e');
    }
  }
}
```

Um pequeno parênteses aqui se você criou o arquivo secrets vai precisar carregar os dados, então na inicialização do seu projeto adicione esse método:

```
Future<void> loadConfig() async {
  final contents = await rootBundle.loadString('secrets.json');
  config = json.decode(contents);
}
```

E agora passe os valores para a classe Logger

```
Logger.initialize(
      endpoint: config?['URL_LOG_API'],
      apiKey: config?['URL_LOG_API_DEV'],
    );
```

Ufa! Tudo configurado.

Basta então enviar nossos logs, e para isso basta criar uma chamada com estrutura próxima ao exemplo e seguindo o padrão pre estabelecido.

No exemplo abaixo trata um log de erro simples, mas lembre-se caso queira enviar o errorStack é importante converter o mesmo em uma lista de strings dessa forma a sua apresentação no resultado ficara igualmente ao log da aplicação.

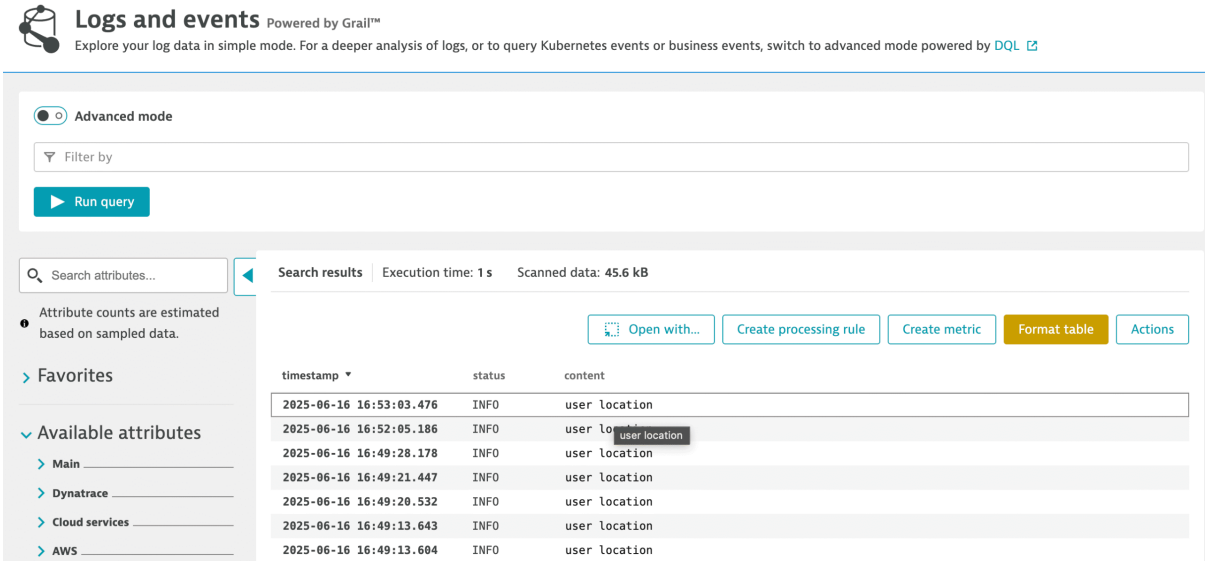
```
Logger.send(Log(
          logType: LogType.TBS,
          acronym: 'smart',
          level: LogLevel.ERROR,
          serviceName: 'smart_recycling',
          namespace: Namespace.prd,
          operation: 'Obter localizaçao',
          content: 'Permissões negadas pelo usuario',
          extra: LogExtraInfo(
            errorMessage: 'Permission not granted',
            exception: e.exception
          )));
```

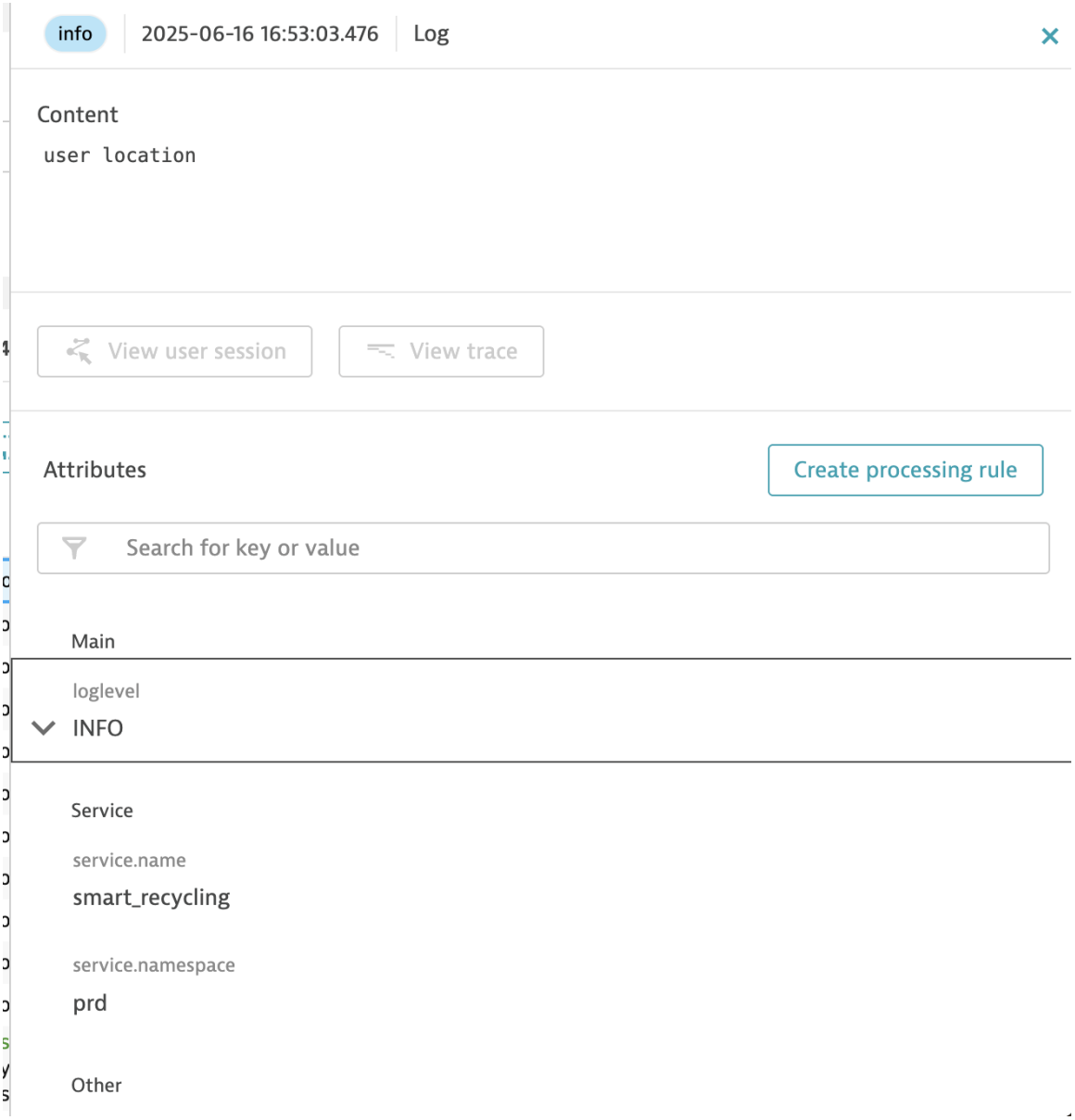
E no próximo exemplo temos um log comum, observem que apenas alteramos e omitimos alguns valores

```
Logger.send(Log(
          logType: LogType.BUS,
          acronym: 'smart',
          level: LogLevel.INFO,
          serviceName: 'smart_recycling',
          namespace: Namespace.prd,
          operation: 'Obter localizaçao',
          content: 'user location',
          extra: LogExtraInfo(
            message: 'Localização obtida com sucesso',
          )));
```

E por fim temos o resultado em nossa ferramenta que está recebendo os dados.

IMPORTANTE: Essa forma que exemplifiquei a vocês é totalmente agnóstica e funciona da mesma forma em qualquer ferramenta de mercado bastando alterar o endpoint para envio dos logs, e dessa forma não corremos o risco de alterar visões e ou estratégias de negócios baseadas em logs uma vez que sempre chegaram da mesma forma.


Logs and events
Powered by Grail™
Explore your log data in simple mode. For a deeper analysis of logs, or to query Kubernetes events or business events, switch to advanced mode powered by DQL
Advanced mode
Filter by
Run query
Search attributes...
Attribute counts are estimated based on sampled data.
Favorites
Available attributes
Main
Dynatrace
Cloud services
AWS
Search results
Execution time: 1 s
Scanned data: 45.6 kB
Open with...
Create processing rule
Create metric
Format table
Actions
timestamp
status
content
2025-06-16 16:53:03.476 INFO user location
2025-06-16 16:52:05.186 INFO user location
2025-06-16 16:49:28.178 INFO user location
2025-06-16 16:49:21.447 INFO user location
2025-06-16 16:49:20.532 INFO user location
2025-06-16 16:49:13.643 INFO user location
2025-06-16 16:49:13.604 INFO user location
user location


info
2025-06-16 16:53:03.476
Log
Content
user location
View user session
View trace
Attributes
Create processing rule
Search for key or value
Main
loglevel
INFO
Service
service.name
smart_recycling
service.namespace
prd
Other