Hoje trago para vocês como realizar a configuração para envio de logs “padronizados”.

Inicialmente essa configuração foi pensada para enviar os logs para o Dynatrace, porem a mesma pode ser aproveitada para outras ferramentas como ElasticSearch, DataDog, etc... Pois foi estruturado um “padrão”.

> O porque do padrão? Simples dessa forma conseguimos ter um log consistente que pode ser processado por diversas ferramentas, sem uso de pipelines adicionais, simplificando a pesquisa, criação de métricas, dashboards e até mesmo o reaproveitamento dos logs para consulta externa por uma API e por fim o principal e mais importante a redução de custos.

No link abaixo temos o repositório da aplicação em que foi desenvolvida, fique a vontade para acompanhar ou fazer o download para facilitar o entendimento.

Link: https://github.com/rafapil/AppFinanceiro

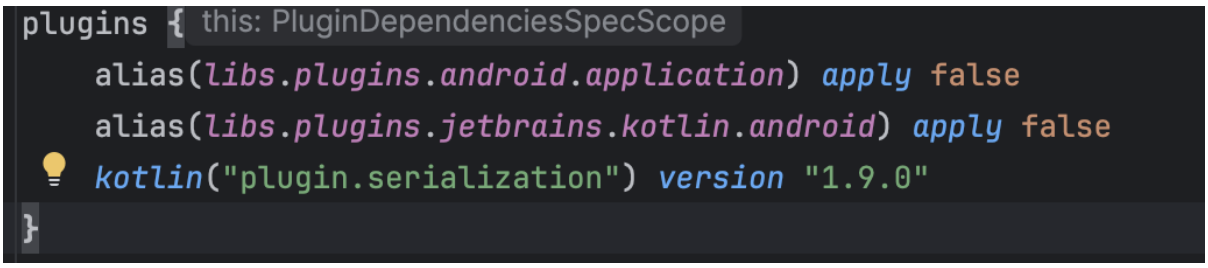
Mas antes de iniciar vamos fazer os imports necessários para o envio de logs

IMPORTANTE: No exemplo a seguir estou usando o Kotlin na versão 1.9.0

No arquivo build.gradle.kts (nível projeto), adicione o valor abaixo para que a serialização funcione corretamente:

```
kotlin("plugin.serialization") version "1.9.0"
```

Exemplo:



Em build.gradle.kts (Module) adicione as seguintes dependências:

```
implementation("com.squareup.okhttp3:okhttp:4.12.0")
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")
implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.3")
```

Opcionalmente você pode definir as dependências no arquivo libs.versions.toml (não obrigatório)

Como estamos focando em padronização vamos iniciar pela criação dos nossos Enums

Por se tratar de um enum que pode ser utilizado em outros momentos, foi adicionado o package logging e dentro dele criamos três enums sendo:

LogLevel:

```
enum class LogLevel(val value: String)  {
    DEBUG("DEBUG"),
    INFO("INFO"),
    WARN("WARN"),
    ERROR("ERROR"),
    FATAL("FATAL")
}
```

LogType:

```
enum class LogType(val value: String) {
    TBS("TBS"), // Troubleshooting
    AUD("AUD"), // Audit
    SEC("SEC"), // Security
    BUS("BUS")  // Business
}
```

Namespace:

```
enum class Namespace(val value: String) {
    PRD("PRD"),
    UAT("UAT"),
    DES("DES")
}
```

Agora precisamos implementar as classes de modelo para definir a padronização de

nossos logs

vamos criar duas classes sendo:

CieloLog

```
import com.projetos.seuprojeto.logging.LogLevel
import com.projetos.seuprojeto.logging.LogType
import com.projetos.seuprojeto.logging.Namespace

data class CieloLog(
    val logType: LogType,
    val acronym: String,
    val level: LogLevel,
    val serviceName: String,
    val namespace: Namespace,
    val operation: String,
    val content: String,
    val duration: Long? = null,
    val value: Double? = null,
    val extra: CieloExtraInfo? = null
)
```

CieloExtraInfo

```
data class CieloExtraInfo(
    val userId: String? = null,
    val transactionId: String? = null,
    val statusCode: Int? = null,
    val exception: String? = null,
    val errorMessage: String? = null,
    val errorStack: List<String>? = null
)
```

Agora que temos a estrutura precisamos criar o objeto que possui as credenciais, aqui é importante falar deixar claro que essa é uma recomendação e caso possua um modelo já estruturado para lidar com credenciais pode asar o mesmo e pular essa parte.

Mas é de suma importância, jamais deixe as credenciais expostas no código ou envie as mesmas para um repositório.

Para o nosso arquivo de credenciais vamos chama-lo de NetworkingConstants

```
object NetworkingConstants {
    const val BASE_URL = "<Sua URL para envio de logs>"
    const val API_KEY = "<Token gerado para enviar logs>"
}
```

Agora vamos criar mais um arquivo o CieloLogExport

Inicializando o arquivo vamos criar inicialmente os data class abaixo:

```
@Serializable
data class LogSerializable(
    val logType: String,
    val acronym: String,
    val logLevel: String,
    @SerialName("service.name") val serviceName: String,
    @SerialName("service.namespace") val namespace: String,
    val operation: String,
    val content: String,
    val duration: Long? = null,
    val value: Double? = null,
    val extra: ExtraInfoDto? = null
)

@Serializable
data class ExtraInfoDto(
    val userId: String? = null,
    val transactionId: String? = null,
    val statusCode: Int? = null,
    val exception: String? = null,
    val errorMessage: String? = null,
    val errorStack: List<String>? = null
)
```

Vamos adicionar a parte que recebe as credenciais para realizar o envio dos logs:

```
private val client = OkHttpClient()
private const val API_URL = NetworkingConstants.BASE_URL
private const val API_TOKEN = "Api-Token ${NetworkingConstants.API_KEY}"
private val json = Json { encodeDefaults = false; ignoreUnknownKeys = true }
```

IMPORTANTE: Por esse exemplo se tratar de um Aplicativo Enviando Logs, no

exemplo adicionei a parte a seguir para utilizar um buffer, logo esta parte é opcional porem recomendada.

Seguindo vamos implementar o buffer:

```
    // Canal com buffer ilimitado (pode trocar por BUFFERED ou conflated)
    private val logChannel = Channel<CieloLog>(Channel.UNLIMITED)

    init {
        // Inicializa o consumo da fila assim que o app sobe
        CoroutineScope(Dispatchers.IO).launch {
            for (log in logChannel) {
                sendLogWithRetry(log)
            }
        }
    }

    fun log(log: CieloLog) {
        CoroutineScope(Dispatchers.IO).launch {
            logChannel.send(log)
        }
    }
```

E agora vamos adicionar a parte responsável por montar a estrutura do log:

```
    private suspend fun sendLogWithRetry(log: CieloLog, tentativas: Int = 3) {
        val logSerializable = LogSerializable(
            logType = log.logType.name,
            acronym = log.acronym,
            logLevel = log.level.name,
            serviceName = log.serviceName,
            namespace = log.namespace.name,
            operation = log.operation,
            content = log.content,
            duration = log.duration,
            value = log.value,
            extra = log.extra?.let {
                ExtraInfoDto(
                    userId = it.userId,
                    transactionId = it.transactionId,
                    statusCode = it.statusCode,
                    exception = it.exception,
                    errorMessage = it.errorMessage,
```

```
                    errorStack = it.errorStack
                )
            }
        )
```

Por fim vamos estruturar o trecho responsável por enviar o log para a API

```
        val requestBody = json.encodeToString(listOf(logSerializable))
            .toRequestBody("application/json; charset=utf-8".toMediaType())

        val request = Request.Builder()
            .url(API_URL)
            .addHeader("Authorization", API_TOKEN)
            .addHeader("Content-Type", "application/json; charset=utf-8")
            .post(requestBody)
            .build()
```

E pra encerrar essa parte o trecho que cuida do retry, lembrando mais uma vez que essa parte é opcional tanto que apenas deixo comentado algumas sugestões que podem ser implementadas.

```
repeat(tentativas) { tentativa ->
            try {
                client.newCall(request).execute().use { response ->
                    if (response.isSuccessful) {
                        println("Log enviado com sucesso")
                        return
                    } else {
                        println("Tentativa ${tentativa + 1}: erro HTTP
${response.code}")
                    }
                }
            } catch (e: Exception) {
                println("Tentativa ${tentativa + 1} falhou:
${e.localizedMessage}")
            }

            // Aguarda um pouco antes da próxima tentativa
            kotlinx.coroutines.delay(1000L * (tentativa + 1))
        }

        // Fallback final: salva localmente ou imprime (pode gravar em arquivo
ou Room aqui)
```

```
        salvarLogLocal(log)
    }

    private fun salvarLogLocal(log: CieloLog) {
        println("Fallback: salvando log local -> $log")
        // Aqui você pode salvar no SharedPreferences, SQLite, Room, arquivo
etc.
    }
```

Caso queira ver a classe completa vou deixar um link do Github Gist

Bem mas ainda falta alguns pontos e um deles e dar permissão de acesso INTERNET no AndroidManifest

Então adicione:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Opcionalmente é possível adicionar um arquivo para permitir o acesso ao endpoint de logs e referenciar o mesmo no Manifest conforme exemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">SUA_URL_PARA_ENVIO_DE_LOGS</domain>
    </domain-config>
</network-security-config>
```

AndroidManifest:

```
<application
    android:allowBackup="true"
    android:networkSecurityConfig="@xml/network_security_config"
```

Pronto! Agora podemos usar a nossa instrumentação para enviar os logs

Neste ponto criei duas funções simples, um para envio de log de eventos, Exemplo: usuário efetuou o pagamento, houve um acesso a funcionalidade xpto, o tempo de execução total foi de 600ms.

A primeira função é o: sendDataLog, que é voltada para o exemplo citado acima!

```
    private fun sendDataLog() {
        CieloLogExporter.log(
            CieloLog(
                logType = LogType.TBS,
                acronym = "xpto",
                level = LogLevel.WARN,
                serviceName = "AppFinanceiro",
                namespace = Namespace.PRD,
                operation = "MainActivity",
                content = "Falha ao buscar saldo",
                duration = 617,
                value = 250.99,
                extra = CieloExtraInfo(
                    userId = "user-001",
                    transactionId = "txn-987654",
                    statusCode = 404,
                    exception = "Error in application",
                    errorMessage = "fail to connect",
                    errorStack = listOf()
                )
            )
        )
    }
```

A segunda função é o sendDataLogError, essa já recebe um parâmetro error do tipo Exception que é usada para detalhar os dados da falha no log; observem que dentro dela ocorre uma conversão do Array<strackTrace> para List<String> isso permite criar uma lista de todos os eventos gerados de forma estruturada no log.

(Exemplo logo abaixo)

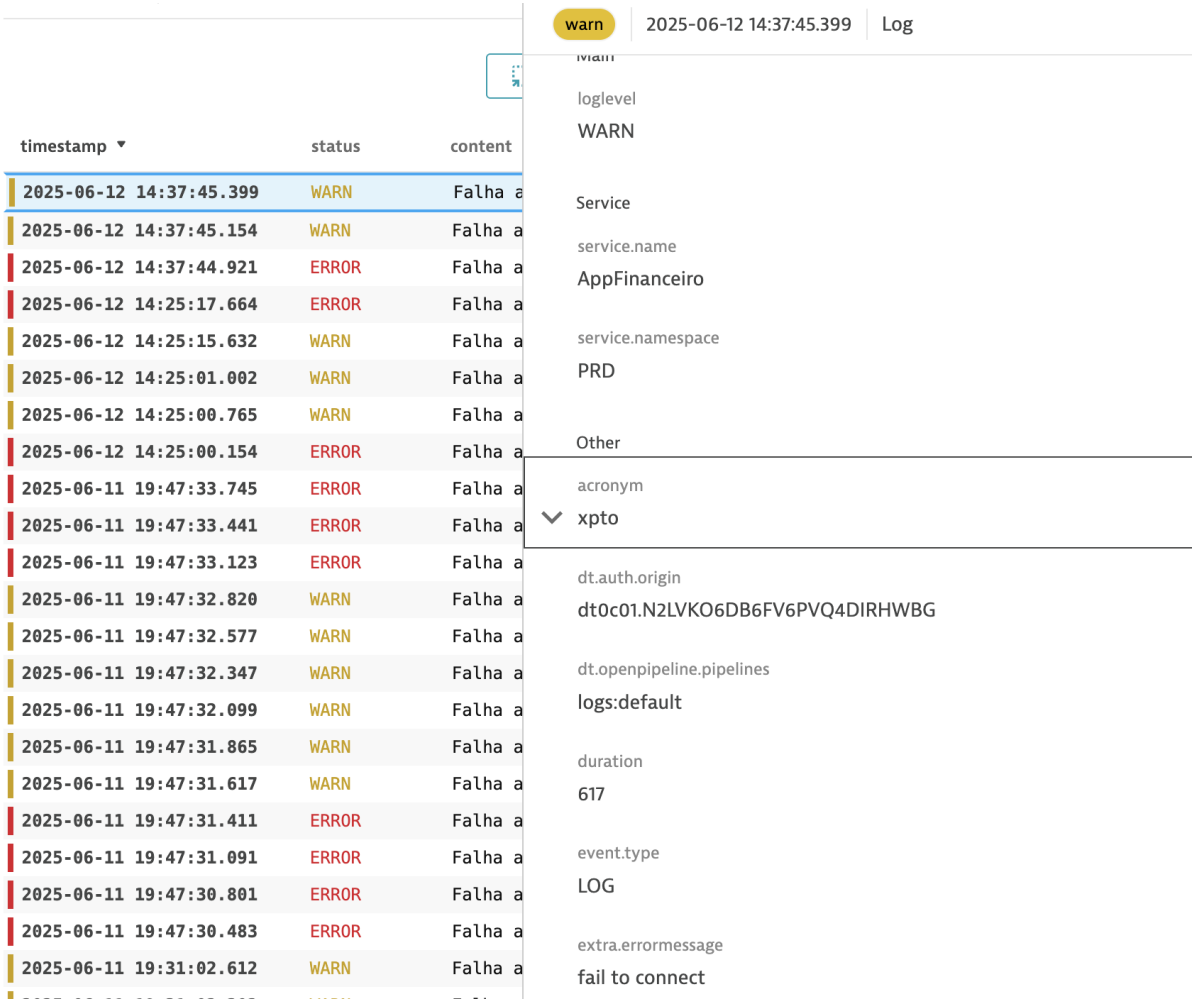
```
    private fun sendDataLogError(error: Exception) {
        val stackErrorString: List<String> = error.stackTrace.map {
it.toString() }
        CieloLogExporter.log(
            CieloLog(
                logType = LogType.TBS,
                acronym = "xpto",
                level = LogLevel.ERROR,
                serviceName = "AppFinanceiro",
                namespace = Namespace.PRD,
```

```
                operation = "MainActivity",
                content = "Falha ao buscar saldo",
                duration = 617,
                value = 250.99,
                extra = CieloExtraInfo(
                    userId = "user-001",
                    transactionId = "txn-987654",
                    statusCode = 404,
                    exception = "Error in application",
                    errorMessage = error.message,
                    errorStack = stackErrorString
                )
            )
        )
    }
```

Para forçar o erro foi adicionado um Throw em uma chamada de botão:

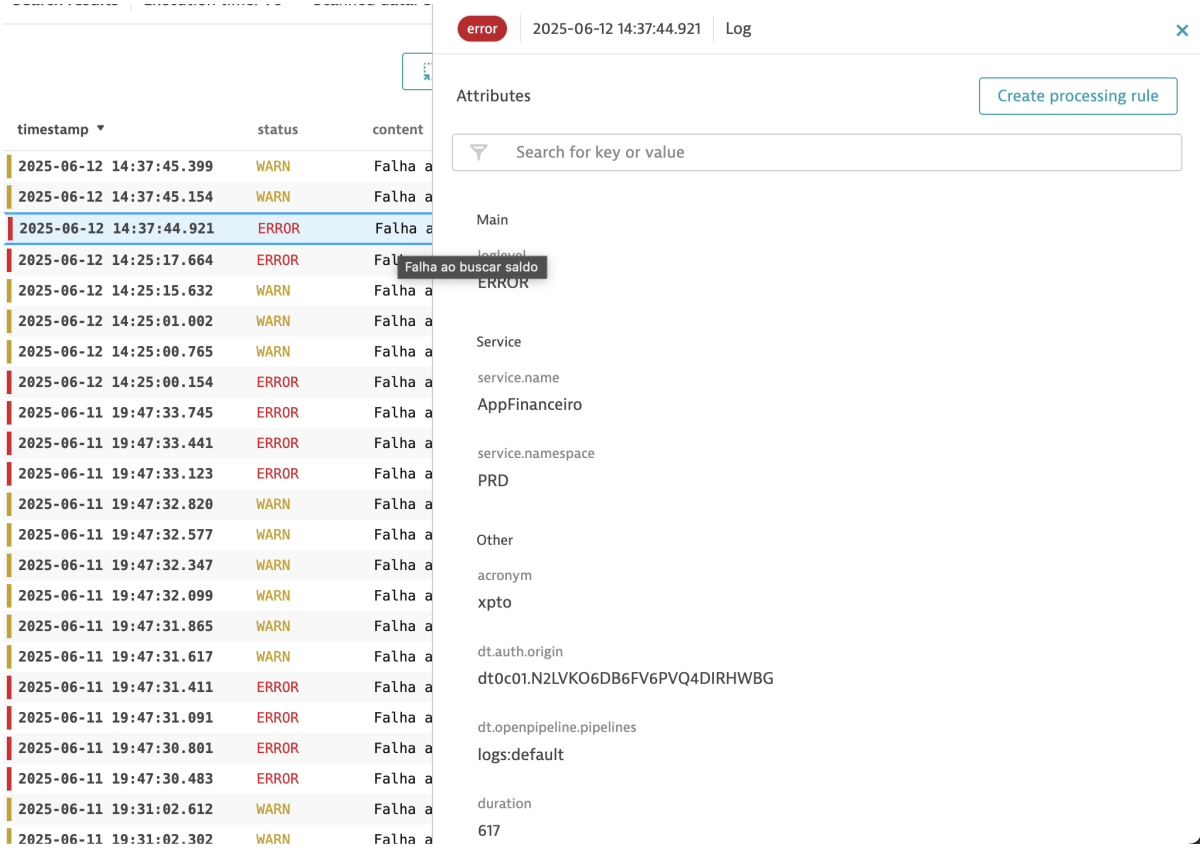
[Segue Github Gist com código completo]

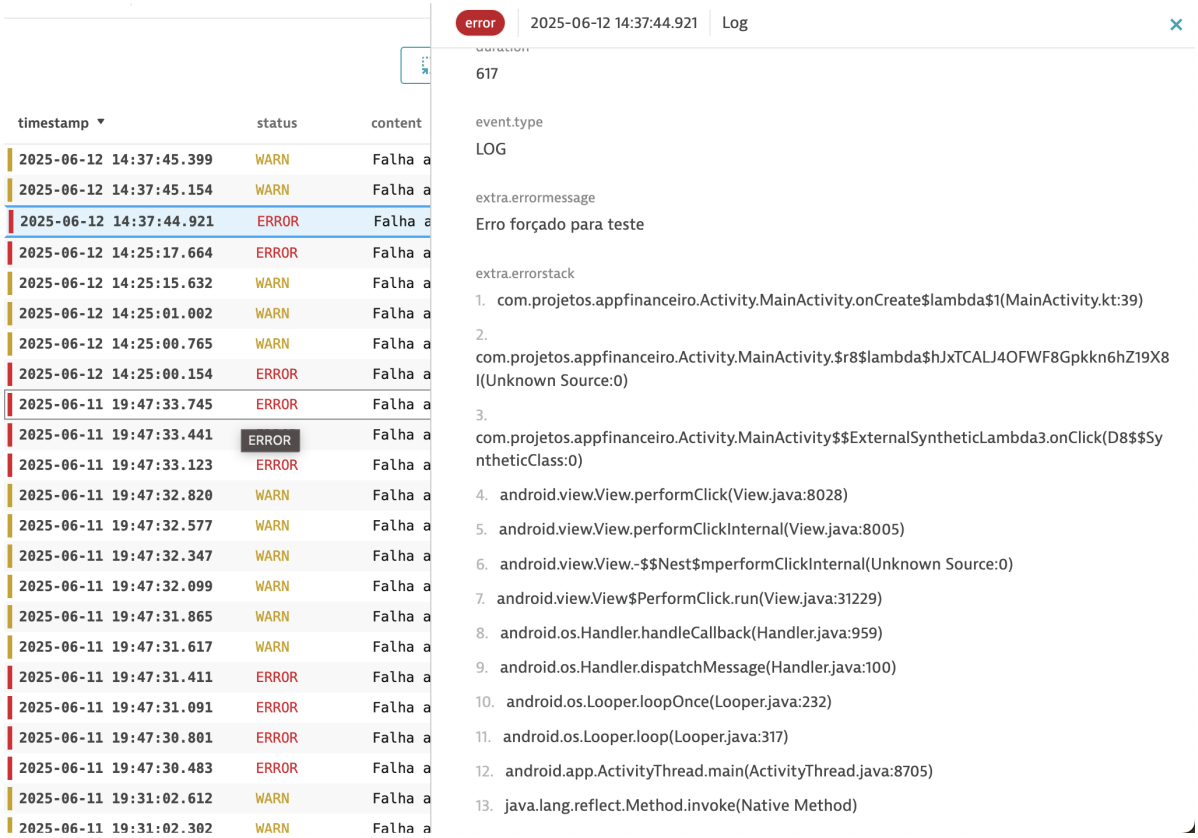
Exemplo dos resultados:
Log de Eventos:



Log de Erro:

Observem a estrutura do stackTrace no log


timestamp
status
content
2025-06-12 14:37:45.399 WARN Falha a
2025-06-12 14:37:45.154 WARN Falha a
2025-06-12 14:37:44.921 ERROR Falha a
2025-06-12 14:25:17.664 ERROR Fal
Falha ao buscar saldo
2025-06-12 14:25:15.632 WARN Falha a
2025-06-12 14:25:01.002 WARN Falha a
2025-06-12 14:25:00.765 WARN Falha a
2025-06-12 14:25:00.154 ERROR Falha a
2025-06-11 19:47:33.745 ERROR Falha a
2025-06-11 19:47:33.441 ERROR Falha a
2025-06-11 19:47:33.123 ERROR Falha a
2025-06-11 19:47:32.820 WARN Falha a
2025-06-11 19:47:32.577 WARN Falha a
2025-06-11 19:47:32.347 WARN Falha a
2025-06-11 19:47:32.099 WARN Falha a
2025-06-11 19:47:31.865 WARN Falha a
2025-06-11 19:47:31.617 WARN Falha a
2025-06-11 19:47:31.411 ERROR Falha a
2025-06-11 19:47:31.091 ERROR Falha a
2025-06-11 19:47:30.801 ERROR Falha a
2025-06-11 19:47:30.483 ERROR Falha a
2025-06-11 19:31:02.612 WARN Falha a
2025-06-11 19:31:02.302 WARN Falha a
error
2025-06-12 14:37:44.921
Log
Attributes
Create processing rule
Search for key or value
Main
ERROR
Service
service.name
AppFinanceiro
service.namespace
PRD
Other
acronym
xpto
dt.auth.origin
dt0c01.N2LVKO6DB6FV6PVQ4DIRHWBG
dt.openpipeline.pipelines
logs:default
duration
617



Dessa forma conseguimos estruturar o log e o envio de forma agnóstica para nosso App

Segue condigo completo para referencia aqui.