Руководство прикладного разработчика компонента Журналирование (Код компонента: LOGA) продукта Platform V Monitor (Код продукта: OPM)

ОГЛАВЛЕНИЕ

Руководство прикладного разработчика

Термины и определения

Общие термины и определения, используемые в данном документе, представлены в общей документации продукта Platform V Monitor (OPM) в документе «Общее описание продукта Platform V Monitor (OPM)».

Ниже приведены специальные термины для компонента Журналирование (LOGA) продукта Platform V Monitor (OPM).

Термин/определение	Определение
Abyss	Сервис Platform V Monitor, предназначен для приема, пред обработки, хранения и получения загруженных данных
Apache Maven	Инструмент автоматизации сборки и управления зависимостями, используемый в Java разработке
FluentBit Sidecar	Инструмент для сбора логов и различных данных телеметрии, является частью компонента LOGA

Системные требования

- компьютер с установленной средой разработки, например, Apache NetBeans
- Apache Maven, версии не ниже 3.6.0;
- Open JDK не ниже версии 11;
- доступ в интернет.

Подключение и конфигурирование

Журналирование

1. Для подключения сервиса журналирования используется FluentBit Sidecar, ниже описаны примеры его конфигураций для Kubernetes (k8s)

Конфигурацию logback, необходимо оформить в виде configMap. Пример конфигурации:

```
apiVersion: "v1" kind: "ConfigMap" metadata: name:
"logback-xml" data: logback.xml: |- <configuration
debug="true"> <property scope="context"
resource="env.properties"/> <appender name="JSON"
class="ch.qos.logback.core.rolling.RollingFileAppende
r"> <encoder
class="net.logstash.logback.encoder.LoggingEventCompo
siteJsonEncoder"> <providers> <timestamp>
```

```
<fieldName>serverEventDatetime</fieldName>
<pattern>[UNIX_TIMESTAMP_AS_NUMBER]</pattern>
</timestamp> <logLevel> <fieldName>level</fieldName>
</logLevel> <loggerName>
<fieldName>logger</fieldName> </loggerName>
<threadName> <fieldName>threadName</fieldName>
</threadName> <message/> <stackTrace>
<fieldName>stackTrace</fieldName> <throwableConverter
class="net.logstash.logback.stacktrace.ShortenedThrow
ableConverter">
<maxDepthPerThrowable>30</maxDepthPerThrowable>
<maxLength>4096</maxLength>
<rootCauseFirst>true</rootCauseFirst>
</throwableConverter> </stackTrace> <mdc>
<excludeMdcKeyName>serverEventDatetime</excludeMdcKey
Name> <excludeMdcKeyName>level</excludeMdcKeyName>
<excludeMdcKeyName>logger</excludeMdcKeyName>
<excludeMdcKeyName>threadName</excludeMdcKeyName>
</mdc> </providers> </encoder> <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRo
llingPolicy"> <fileNamePattern>/fluent-
bit/etc/logs/log_%d{yyyy-MM-
dd}_%i.json</fileNamePattern>
<maxFileSize>5MB</maxFileSize>
<maxHistory>5</maxHistory>
<totalSizeCap>50MB</totalSizeCap> </rollingPolicy>
</appender> <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
<encoder> <pattern>%date %level [%thread] %logger{5}
- %msg %n %xEx</pattern> <charset>UTF-8</charset>
</encoder> </appender> <root level="INFO"> <appender-
ref ref="STDOUT"/> <appender-ref ref="JSON"/> </root>
</configuration>
```

2. Создать конфигурации FluentBit Sidecar в виде configMap. Пример конфигурации:

```
apiVersion: v1 kind: ConfigMap metadata: name:
volume-conf-fluent-bit-sidecar data: fluent-bit.conf:
|- [SERVICE] Flush 1 Daemon Off Parsers_File /fluent-
bit/etc/parsers.conf HTTP_Server On HTTP_Listen
0.0.0.0 HTTP_PORT 9081 Log_Level info [INPUT] Name
tail Tag file.tail Path /fluent-bit/etc/logs/*.json
Mem_Buf_Limit 10MB Skip_Long_Lines On
Refresh_Interval 2 Rotate_Wait 1 Read_from_Head Off
```

```
DB /fluent-bit/etc/logs/kube.db Parser custom
[FILTER] Name modify Match * Add hostname ${HOSTNAME}
[FILTER] Name modify Match * Add tenant {{
TENANT_CODE }} [FILTER] Name modify Match * Add ufs-
logger-appFQDN ${ufs-logger.server.appFQDN} [FILTER]
Name modify Match * Add ufs-logger-http.appFQDN
${ufs-logger-http.server.appFQDN} [OUTPUT] Name kafka
Match file.tail Brokers ${ufs-
logger.kafka.bootstrap.servers} Topics ${ufs-
logger.kafka.topic} rdkafka.security.protocol ${ufs-
logger.kafka.security.protocol}
rdkafka.ssl.key.password ${rdkafka.ssl.key.password}
rdkafka.ssl.key.location
/etc/config/ssl/logger_private-key.pem
rdkafka.ssl.certificate.location
/etc/config/ssl/logger_cert.pem
rdkafka.ssl.ca.location
/etc/config/ssl/logger_cacerts.cer rdkafka.log_level
6 rdkafka.queue.buffering.max.kbytes 5120
parsers.conf: |- [PARSER] Name custom Format json
```

3. В deploymentConfig создать разделы и примонтировать данные configMap:

```
apiVersion: apps/v1 kind: Deployment … spec: …
template: … spec: containers: … - envFrom:
volumeMounts: - mountPath: '/opt/logger-pl/conf'
name: logback-xml - mountPath: '/fluent-bit/etc/logs'
name: logsshare volumes: - name: logback-xml
configMap: name: logback-xml defaultMode: 420 - name:
logsshare emptyDir: {} - name: volume-conf-fluent-
bit-sidecar configMap: name: volume-conf-fluent-bit-
sidecar defaultMode: 420
```

4. Указать в deploymentConfig, чтобы к подам устанавливался FluentBit Sidecar:

```
apiVersion: apps/v1 kind: Deployment metadata: …
spec: … template: … spec: containers: … - name:
fluent-bit-sidecar image:
'${dockerRegistry}/ci01976100/ci02698091_ulogger/flue
nt-
bit@sha256:e38955b3495fe6a40b0506326c1d8458e2f03ad677
47406af75f3719678b6036' resources: limits: cpu:
'${logger-pl.cpuLimit}' memory: '${logger-
```

```
pl.memLimit}' requests: cpu: '${logger-
pl.cpuRequest}' memory: '${logger-pl.memRequest}'
env: - name: pod valueFrom: fieldRef: appVersion: v1
fieldPath: metadata.name volumeMounts: - mountPath:
'/fluent-bit/etc/logs' name: logsshare - mountPath:
'/fluent-bit/etc' name: volume-conf-fluent-bit-
sidecar - mountPath: '/etc/config/ssl' name: kafka-
cert-vol ports: - name: metrics-9081 protocol: TCP
containerPort: 9081 dnsPolicy: ClusterFirst
```

5. В скрипте запуска своего приложения указать в аргументах запуска путь к конфигурации логгера logback:

```
java \

-Dlogback.configurationFile="/opt/logger-
pl/conf/logback.xml" \

-jar /opt/demo.jar
```

Трейсинг

Для подключения трейсинга необходимо добавить в свое приложение следующие зависимости (показан пример для web-приложения с использованием springboot): **pom.xml**

```
<dependencies> <dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency> <dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope> </dependency> <!--зависимости для
zipkin--> <dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-zipkin</artifactId>
<version>2.2.7.RELEASE</version> </dependency>
<dependency> <groupId>p6spy</groupId>
<artifactId>p6spy</artifactId>
<version>3.9.1</version> </dependency> <dependency>
<groupId>io.zipkin.brave</groupId> <artifactId>brave-
instrumentation-p6spy</artifactId>
<version>5.13.3</version> </dependency>
</dependencies>
```

Так же добавить properties и в секции dependencyManagement указать: **pom.xml**

```
<properties> <java.version>11</java.version>
<spring.cloud-version>2020.0.2</spring.cloud-version>
</properties> <dependencyManagement> <dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring.cloud-version}</version>
<type>pom</type> <scope>import</scope> </dependency>
</dependencies> </dependencyManagement>
```

В файле application.properties или application.yaml указать строку подключения к сервису трейсинга: **application.properties**

```
spring.zipkin.base-url=http://адрес-сервиса-
трассировки/trace
```

Миграция на текущую версию

Миграция сервиса не предусмотрена.

Разработка первого приложения с использованием программного продукта

Быстрый старт

Журналирование

1. Сгенерировать новый проект, например, с сайта https://start.spring.io/ (используя все настройки по умолчанию).
2. В pom.xml добавить зависимости в секции dependencies, указанные в разделе «Подключение и конфигурирование». Должен получиться файл со следующим содержанием:

```
<?xml version="1.0" encoding="UTF-8"?> <project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion> <parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.4.5</version> <relativePath/> <!-- lookup
```

```
parent from repository --> </parent>
<groupId>com.example</groupId>
<artifactId>demo</artifactId> <version>0.0.1-
SNAPSHOT</version> <name>demo</name>
<description>Demo project for Spring
Boot</description> <properties>
<java.version>11</java.version> </properties>
<dependencies> <dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency> <!-- зависимости для журналирования -->
<dependency> <groupId>ch.qos.logback</groupId>
<artifactId>logback-classic</artifactId>
<version>1.2.3</version> </dependency> <dependency>
<groupId>net.logstash.logback</groupId>
<artifactId>logstash-logback-encoder</artifactId>
<version>6.3</version> </dependency> <dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope> </dependency> </dependencies>
<build> <plugins> <plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin> </plugins> </build> </project>
```

3. Открыть класс DemoApplication и объявить в нем private static final переменную типа Logger из пакета slf4j и проинициализировать её при помощи LoggerFactory того же пакета:

```
public static final Logger LOGGER =
LoggerFactory.getLogger(DemoApplication.class);
```

4. Из метода main данного класса вызвать логгер с уровнем info, например, и сообщением «!!!!!!! Привет Мир !!!!!!!». Метод main теперь будет выглядеть так:

```
public static void main(String[] args) {
SpringApplication.run(DemoApplication.class, args);
LOGGER.info("!!!!!!! Привет Мир !!!!!!!"); }
```

5. Теперь необходимо сконфигурировать логгер, для этого добавим в папку с ресурсами проекта src/main/resources файл с именем logback.xml и следующим содержимым:

```
<configuration debug="true"> <property
scope="context" resource="application.properties"/>
<appender name="JSON"
class="ch.qos.logback.core.rolling.RollingFileAppende
r"> <encoder
class="net.logstash.logback.encoder.LoggingEventCompo
siteJsonEncoder"> <providers> <timestamp>
<fieldName>serverEventDatetime</fieldName>
<pattern>[UNIX_TIMESTAMP_AS_NUMBER]</pattern>
</timestamp> <logLevel> <fieldName>level</fieldName>
</logLevel> <loggerName>
<fieldName>logger</fieldName> </loggerName>
<threadName> <fieldName>threadName</fieldName>
</threadName> <message/> <stackTrace>
<fieldName>stackTrace</fieldName> <throwableConverter
class="net.logstash.logback.stacktrace.ShortenedThrow
ableConverter">
<maxDepthPerThrowable>30</maxDepthPerThrowable>
<maxLength>4096</maxLength>
<rootCauseFirst>true</rootCauseFirst>
</throwableConverter> </stackTrace> <pattern>
<omitEmptyFields>true</omitEmptyFields> <pattern> {
"application": "${spring.application.name}" }
</pattern> </pattern> <mdc>
<excludeMdcKeyName>serverEventDatetime</excludeMdcKey
Name> <excludeMdcKeyName>level</excludeMdcKeyName>
<excludeMdcKeyName>logger</excludeMdcKeyName>
<excludeMdcKeyName>threadName</excludeMdcKeyName>
</mdc> </providers> </encoder> <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRo
llingPolicy"> <fileNamePattern>logs/log_%d{yyyy-MM-
dd}_%i.json</fileNamePattern>
<maxFileSize>5MB</maxFileSize>
<maxHistory>5</maxHistory>
<totalSizeCap>50MB</totalSizeCap> </rollingPolicy>
</appender> <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
<encoder> <pattern>%date %level [%thread] %logger{5}
- %msg %n %xEx</pattern> <charset>UTF-8</charset>
</encoder> </appender> <root level="INFO"> <appender-
ref ref="STDOUT"/> <appender-ref ref="JSON"/> </root>
</configuration>
```

Здесь Logback сконфигурирован с двумя appender, STDOUT и JSON. STDOUT позволяет увидеть логи в консоли Kubernetes или терминале разработчика, при отладке локально. JSON – это файловый appender, который пишет логи в формате JSON, что удобно для дальнейшей обработки и записи в различные системы (Kafka, Elasticsearch, etc.). В качестве json-encoder в appender используется реализация от Logstash - .

Он позволяет обогащать логи различной информацией, в том числе содержимым MDC-контекста логгера. Так что по желанию логи могут быть обогащены нужными дополнительными атрибутами, для этого надо изменить или дополнить конфигурацию LoggingEventCompositeJsonEncoder.

Так же можно добавлять в записи логов статичные атрибуты приложения, для их идентификации. Данные атрибуты указываются в секции pattern:

```
pattern>     <omitEmptyFields>true</omitEmptyFields>
<pattern>         {         "application":
"${spring.application.name}"         }     </pattern>
</pattern>
```

Откроем файл application.properties из папки проекта src/main/resources и добавим туда строку:

```
spring.application.name=DemoApplication
```

Аналогичным способом можно добавлять и другие атрибуты, например, версию приложения и т.п.

Логи пишутся последовательно в несколько файлов, что обеспечивает их плавную обработку FluentBit, минимизирует потери логов при большой интенсивности записи. По мере заполнения, файлы ротируются, старые постепенно удаляются, пишутся новые с другими номерами, по принципу очереди.

Политика ротации настраивается следующим образом:

```
<rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRo
llingPolicy">     <fileNamePattern>logs/log_%d{yyyy-
MM-dd}_%i.json</fileNamePattern>
<maxFileSize>5MB</maxFileSize>
<maxHistory>5</maxHistory>
<totalSizeCap>50MB</totalSizeCap> </rollingPolicy>
```

,где: %i - порядковый номер файла,

%d{yyyy-MM-dd} - дата создания файла,

maxFileSize - максимальный размер файла, по достижению которого создается новый файл, а старый в последствии удалится, по мере накопления общего объема всех файлов логов,

установленного в totalSizeCap. Здесь в примере будет создано и все время поддерживаться 10 файлов.
При гибкой настройке этих параметров, можно минимально нивелировать потерю их FluentBit.

6. Можно запускать приложение, в результате, перед завершением приложения, увидим в логе запись:

```
2021-05-04 16:38:43,756 INFO [main]
c.e.d.DemoApplication - !!!!!!! Привет Мир !!!!!!!
```

Так же будет создан файл в папке проекта logs, с логами в формате JSON.

Трейсинг

1. Сгенерировать новый проект, например, с сайта https://start.spring.io/ (используя все настройки по умолчанию).
2. В pom.xml добавить зависимости в секции dependencies, указанные в разделе «Подключение и конфигурирование». Должен получиться файл со следующим содержанием:

```
<?xml version="1.0" encoding="UTF-8"?> <project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>     <parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.4.4</version>         <relativePath/> <!--
lookup parent from repository -->     </parent>
<groupId>ru.example</groupId>
<artifactId>demo</artifactId>     <version>0.0.1-
SNAPSHOT</version>     <name>demo</name>
<description>Demo project for Spring
Boot</description>     <properties>
<java.version>11</java.version>
<spring.cloud-version>2020.0.2</spring.cloud-version>
</properties>     <dependencies>         <dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>           <dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>         </dependency>
<!--зависимости zipkin-->         <dependency>
```

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-zipkin</artifactId>
<version>2.2.7.RELEASE</version>
</dependency>           <dependency>
<groupId>p6spy</groupId>
<artifactId>p6spy</artifactId>
<version>3.9.1</version>        </dependency>
<dependency>
<groupId>io.zipkin.brave</groupId>
<artifactId>brave-instrumentation-p6spy</artifactId>
<version>5.13.3</version>         </dependency>
</dependencies>       <build>         <plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>         </plugins>     </build>
<dependencyManagement>         <dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring.cloud-version}</version>
<type>pom</type>
<scope>import</scope>             </dependency>
</dependencies>     </dependencyManagement>
</project>
```

3. Добавить REST-контроллер, с двумя методами: **ZipkinRestController.java**

```
package ru.example.demo.controller;     import
org.slf4j.Logger; import org.slf4j.LoggerFactory;
import
org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.RestControlle
r;   @RestController public class
ZipkinRestController {     private static final
Logger LOGGER =
LoggerFactory.getLogger(ZipkinRestController.class);
@GetMapping("/rest1")     public String rest1(){
LOGGER.info("Вызван метод rest1");         return
"OK";     }       @GetMapping("/rest2")     public
String rest2(){         LOGGER.info("Вызван метод
rest2");         return "OK";     }
@GetMapping("/rest3")     public String rest3(){
```

```
LOGGER.info("Вызван метод rest3");          return
"OK";      } }
```

4. В файл application.propeties добавить строку для REST-сервиса трейсинга: **application.properties**

```
spring.zipkin.base-url=http://unver.tracing-
collector.ci01976100-edevgen-log1-dev.apps.dev-
gen.sigma.sbrf.ru/trace
```

5. Запустить приложение.
6. Теперь при вызове методов /rest1, /rest2 и /rest3 приложение будет отправлять спаны. Откройте в браузере адрес http://localhost:8080/rest1, в результате в трейсинг отправится спан с таким содержимым: **пример спана**

```
{     "traceId": "7c10e61aa390c96e",      "parentId":
null,     "id": "7c10e61aa390c96e",     "kind":
"SERVER",     "name": "get /rest1",     "timestamp":
1627571927993,     "duration": 2172,
"localServiceName": "default",     "local_ipv4":
"10.6.21.179",     "remoteServiceName": null,
"tag_http.method": "GET",     "tag_http.path":
"/rest1",     "tag_mvc.controller.class":
"ZipkinRestController",
"tag_mvc.controller.method": "rest1" }
```

7. Теперь усложним приложение, чтобы увидеть ветвления спанов. Для этого подключим следующие артефакты: **pom.xml**

```
<!--для работы с кафкой--> <dependency>
<groupId>org.springframework.kafka</groupId>
<artifactId>spring-kafka</artifactId>
<version>2.6.7</version> </dependency> <!--для работы
с jdbc--> <dependency>
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId> </dependency>
<dependency>     <groupId>com.zaxxer</groupId>
<artifactId>HikariCP</artifactId> </dependency>
```

8. Добавим java-конфиг **ZipkinAppConfig.java**

```
package ru.example.demo.configuration;   import
com.p6spy.engine.spy.P6DataSource; import
com.zaxxer.hikari.HikariConfig; import
com.zaxxer.hikari.HikariDataSource; import
org.springframework.context.annotation.Bean; import
org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;
import javax.sql.DataSource;   @Configuration public
class ZipkinAppConfig {       @Bean     RestTemplate
getRestTemplate() {         return new
RestTemplate();     }         @Bean     DataSource
getDataSource() {         HikariConfig hikariConfig =
new HikariConfig();
hikariConfig.setDriverClassName("org.postgresql.Drive
r");
hikariConfig.setJdbcUrl("jdbc:postgresql://addr:5432/
logging");         hikariConfig.setUsername("login");
hikariConfig.setPassword("pass");
hikariConfig.setPoolName("hikariPool");
HikariDataSource dataSource = new
HikariDataSource(hikariConfig);           return new
P6DataSource(dataSource);     }     }
```

В REST-контроллере объявим три bean:

1. DataSource - для работы с БД.
2. RestTemplate - для http-вызовов.
3. KafkaTemplate - для записи в Kafka.

Напишем в каждом из методов вызовы для них.

1. Код должен получиться примерно такой: **ZipkinRestController.java**

```
package ru.example.demo.controller;     import
org.slf4j.Logger; import org.slf4j.LoggerFactory;
import
org.springframework.beans.factory.annotation.Autowire
d; import
org.springframework.kafka.core.KafkaTemplate; import
org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.RestControlle
r; import
org.springframework.web.client.RestTemplate;   import
javax.sql.DataSource; import java.sql.Connection;
```

```
import java.sql.PreparedStatement; import
java.sql.ResultSet; import java.sql.SQLException;
@RestController public class ZipkinRestController {
private static final Logger LOGGER =
LoggerFactory.getLogger(ZipkinRestController.class);
@Autowired     public DataSource dataSource;
@Autowired     public RestTemplate restTemplate;
@Autowired     private KafkaTemplate<Object, Object>
kafkaTemplate;       @GetMapping("/rest1")     public
String rest1(){         LOGGER.info("Вызван метод
rest1");         try (                Connection
connection = dataSource.getConnection();
PreparedStatement statement =
connection.prepareStatement("SELECT * FROM
property");                 ResultSet result =
statement.executeQuery()                ){
if (result.next()){
LOGGER.info("Found some record");             }
else {                LOGGER.info("No records
found");             }         }         catch
(SQLException ex){             ex.printStackTrace();
}         return "OK";     }
@GetMapping("/rest2")     public String rest2(){
LOGGER.info("Вызван метод rest2");         return
restTemplate.getForObject("https://someaddress.com",S
tring.class);     }       @GetMapping("/rest3")
public String rest3(){         LOGGER.info("Вызван
метод rest3");
kafkaTemplate.send("j2","4321");         return "OK";
} }
```

2. Запускаем наше приложение. Теперь вместо одного спана, при вызове каждого endpoint нашего приложения, оно будет генерировать по два спана (первый - начало трейса, второй - его ответвление на другие сервисы): **Результат вызова метода rest1**


```
{     "traceId": "d74804d217ff4f69",     "parentId":
null,     "id": "d74804d217ff4f69",     "kind":
"SERVER",     "name": "get /rest1",     "timestamp":
1627656668323,     "duration": 44696,
"localServiceName": "default",     "local_ipv4":
"10.6.26.173",     "remoteServiceName": null,
"tag_http.method": "GET",     "tag_http.path":
"/rest1",     "tag_mvc.controller.class":
"ZipkinRestController",
```

"tag_mvc.controller.method": "rest1" } и { "traceId": "d74804d217ff4f69", "parentId": "d74804d217ff4f69", "id": "8c31cc7d47f4df0b", "kind": "CLIENT", "name": "select", "timestamp": 1627656668345, "duration": 19159, "localServiceName": "default", "local_ipv4": "10.6.26.173", "remoteServiceName": "logging", "remote_ipv4": "10.53.88.107", "tag_sql.query": "SELECT * FROM property" }

Результат вызова метода rest2

{ "traceId": "19ca0739ab8a1810", "parentId": null, "id": "19ca0739ab8a1810", "kind": "SERVER", "name": "get /rest2", "timestamp": 1627656885318, "duration": 1645781, "localServiceName": "default", "local_ipv4": "10.6.26.173", "remoteServiceName": null, "tag_error": "Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on GET request for \"https://someaddress.com\": PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target; nested exception is javax.net.ssl.SSLHandshakeException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target", "tag_http.method": "GET", "tag_http.path": "/rest2", "tag_http.status_code": "500", "tag_mvc.controller.class": "ZipkinRestController", "tag_mvc.controller.method": "rest2" } и { "traceId": "19ca0739ab8a1810", "parentId": "19ca0739ab8a1810", "id": "aeac85ecebf78c0a", "kind": "CLIENT", "name": "get", "timestamp": 1627656885349, "duration": 1604125, "localServiceName": "default", "local_ipv4": "10.6.26.173", "remoteServiceName": null, "tag_error": "PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to

```
requested target",     "tag_http.method": "GET",
"tag_http.path": "" }
```


Результат вызова метода rest3


```
{     "traceId": "44c6ef02b7a4b75d",     "parentId":
null,     "id": "44c6ef02b7a4b75d",     "kind":
"SERVER",     "name": "get /rest3",     "timestamp":
1627657059135,     "duration": 666969,
"localServiceName": "default",     "local_ipv4":
"10.6.26.173",     "remoteServiceName": null,
"tag_http.method": "GET",     "tag_http.path":
"/rest3",     "tag_mvc.controller.class":
"ZipkinRestController",
"tag_mvc.controller.method": "rest3" } и { "traceId":
"44c6ef02b7a4b75d", "parentId": "44c6ef02b7a4b75d",
"id": "b8a10f8b991df241", "kind": "PRODUCER", "name":
"send", "timestamp": 1627657059323, "duration":
558053, "localServiceName": "default", "local_ipv4":
"10.6.26.173", "remoteServiceName": "kafka",
"tag_kafka.topic": "j2" }
```


3. Как видно из примера, первые спаны не имеют родительского и в поле kind у них указано значение SERVER, а вторые спаны ссылаются на первые как на родительский и у них kind уже клиентский. В зависимости от типа взаимодействия так же отличается набор тегов спана.

Использование программного продукта

Сервис используется для записи событий журналирования в хранилище, предоставления пользовательского интерфейса для просмотра и поиска этих событий, а также выгрузки в файл.

Кейсы журналирования:

№	Use cases	Шаги

1	Получение диагностики для тестирования или отладки	1. Инженер тестирования выполняет тест-кейсы своего продукта. 2. Анализирует логи на предмет ошибок. 3. При выявлении внутренних или внешних выполняет выгрузку логов. 4. Регистрирует тикет/инцидент и прикрепляет выгруженные логи, для диагностики.
2	Контроль корректного запуска приложения	1. Потребитель выполняет запуск своих приложений. 2. Во время запуска, возникают ошибки, из-за которых старт приложения невозможен. 3. Инженер выгружает логи старта приложения. 4. Регистрирует тикет/инцидент и прикрепляет выгруженные логи для диагностики.

3	Сбор диагностики для обработки претензий пользователей	1. Пользователь сообщает на первую линии поддержки о проблемах. 2. Сотрудник первой линии поддержки идентифицирует поведение как не ожидаемое. 3. Передает данные и пользователе на вторую линию поддержки, для сбора диагностики и идентификации проблемы. 4. Инженер второй линии выгружает логи и передает диагностику разработчикам.

Кейсы трейсинга:

№	Use cases	Шаги

1	Нагрузочное тестирование	1. Запуск нагрузочного тестирования. 2. При выявлении отклонений от желаемых результатов производительности необходимо найти трассировку указанного шага тестирования. 3. Проанализировать на каких узлах (спанах) время выполнения непозволительно велико. 4. Передать информацию для анализа разработке или рекомендации по выполнению повышения производительности каких-либо связанных компонентов.
2	Оптимизация бизнес-процессов	1. Потребитель выполняет тест-кейсы своих бизнес-процессов. 2. Находит их трассировки. 3. Анализирует их на отсутствие избыточных взаимодействий. 4. Принимает решение об исключении избыточных взаимодействий из своих бизнес-процессов или процессов смежных систем.

3	Выявление неработоспособности. Определенных узлов/микросервисов и сбор диагностики	1. Инженер сопровождения при выявлении большого числа неуспешных трассировок идентифицирует узлы в трассировках, на которых происходит ошибка. 2. Осуществляет переход от неуспешных спанов к логам. 3. Анализирует логи. 4. При необходимости выгружает логи и передает разработчикам для определения проблемы

Часто встречающиеся проблемы и пути их устранения

База знаний по эксплуатации решения не накоплена.