Prometheus and Grafana:

Monitoring Linux host metrics with the Node Exporter

* [Installing and running the Node Exporter](https://prometheus.io/docs/guides/node-exporter/#installing-and-running-the-node-exporter)
* [Node Exporter metrics](https://prometheus.io/docs/guides/node-exporter/#node-exporter-metrics)
* [Configuring your Prometheus instances](https://prometheus.io/docs/guides/node-exporter/#configuring-your-prometheus-instances)
* [Exploring Node Exporter metrics through the Prometheus expression browser](https://prometheus.io/docs/guides/node-exporter/#exploring-node-exporter-metrics-through-the-prometheus-expression-browser)

The Prometheus [**Node Exporter**](https://github.com/prometheus/node_exporter) exposes a wide variety of hardware- and kernel-related metrics.

In this guide, you will:

* Start up a Node Exporter on localhost
* Start up a Prometheus instance on localhost that's configured to scrape metrics from the running Node Exporter

**NOTE:** While the Prometheus Node Exporter is for \*nix systems, there is the [Windows exporter](https://github.com/prometheus-community/windows_exporter) for Windows that serves an analogous purpose.

Installing and running the Node Exporter

The Prometheus Node Exporter is a single static binary that you can install [via tarball](https://prometheus.io/docs/guides/node-exporter/#tarball-installation). Once you've downloaded it from the Prometheus [downloads page](https://prometheus.io/download#node_exporter) extract it, and run it:

# NOTE: Replace the URL with one from the above mentioned "downloads" page.

# <VERSION>, <OS>, and <ARCH> are placeholders.

wget https://github.com/prometheus/node\_exporter/releases/download/v<VERSION>/node\_exporter-<VERSION>.<OS>-<ARCH>.tar.gz

tar xvfz node\_exporter-\*.\*-amd64.tar.gz

cd node\_exporter-\*.\*-amd64

./node\_exporter

You should see output like this indicating that the Node Exporter is now running and exposing metrics on port 9100:

INFO[0000] Starting node\_exporter (version=0.16.0, branch=HEAD, revision=d42bd70f4363dced6b77d8fc311ea57b63387e4f) source="node\_exporter.go:82"

INFO[0000] Build context (go=go1.9.6, user=root@a67a9bc13a69, date=20180515-15:53:28) source="node\_exporter.go:83"

INFO[0000] Enabled collectors: source="node\_exporter.go:90"

INFO[0000] - boottime source="node\_exporter.go:97"

...

INFO[0000] Listening on :9100 source="node\_exporter.go:111"

Node Exporter metrics

Once the Node Exporter is installed and running, you can verify that metrics are being exported by cURLing the /metrics endpoint:

curl http://localhost:9100/metrics

You should see output like this:

# HELP go\_gc\_duration\_seconds A summary of the GC invocation durations.

# TYPE go\_gc\_duration\_seconds summary

go\_gc\_duration\_seconds{quantile="0"} 3.8996e-05

go\_gc\_duration\_seconds{quantile="0.25"} 4.5926e-05

go\_gc\_duration\_seconds{quantile="0.5"} 5.846e-05

# etc.

Success! The Node Exporter is now exposing metrics that Prometheus can scrape, including a wide variety of system metrics further down in the output (prefixed with node\_). To view those metrics (along with help and type information):

curl http://localhost:9100/metrics | grep "node\_"

Configuring your Prometheus instances

Your locally running Prometheus instance needs to be properly configured in order to access Node Exporter metrics. The following [prometheus.yml](https://prometheus.io/docs/prometheus/latest/configuration/configuration/) example configuration file will tell the Prometheus instance to scrape, and how frequently, from the Node Exporter via localhost:9100:

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: node

static\_configs:

- targets: ['localhost:9100']

To install Prometheus, [download the latest release](https://prometheus.io/download) for your platform and untar it:

wget https://github.com/prometheus/prometheus/releases/download/v\*/prometheus-\*.\*-amd64.tar.gz

tar xvf prometheus-\*.\*-amd64.tar.gz

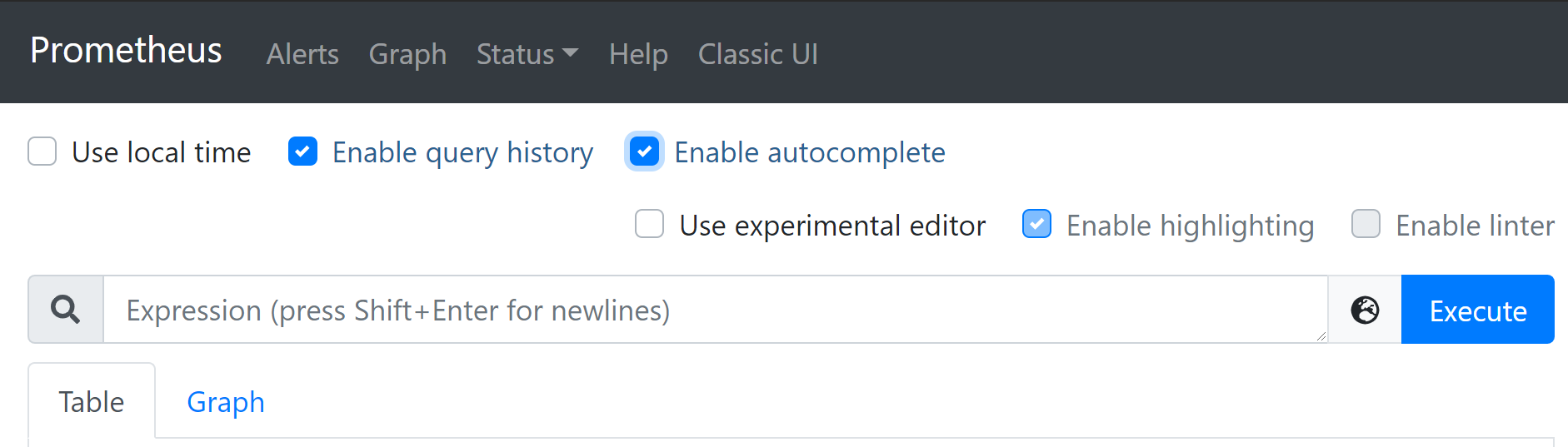
cd prometheus-\*.\*

Once Prometheus is installed you can start it up, using the --config.file flag to point to the Prometheus configuration that you created [above](https://prometheus.io/docs/guides/node-exporter/#config):

./prometheus --config.file=./prometheus.yml

Exploring Node Exporter metrics through the Prometheus expression browser

Now that Prometheus is scraping metrics from a running Node Exporter instance, you can explore those metrics using the Prometheus UI (aka the [expression browser](https://prometheus.io/docs/visualization/browser)). Navigate to localhost:9090/graph in your browser and use the main expression bar at the top of the page to enter expressions. The expression bar looks like this:



Metrics specific to the Node Exporter are prefixed with node\_ and include metrics like node\_cpu\_seconds\_total and node\_exporter\_build\_info.

Click on the links below to see some example metrics:

| **Metric** | **Meaning** |
| --- | --- |
| [rate(node\_cpu\_seconds\_total{mode="system"}[1m])](http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(node_cpu_seconds_total%7Bmode%3D%22system%22%7D%5B1m%5D)&g0.tab=1) | The average amount of CPU time spent in system mode, per second,  over the last minute (in seconds) |
| [node\_filesystem\_avail\_bytes](http://localhost:9090/graph?g0.range_input=1h&g0.expr=node_filesystem_avail_bytes&g0.tab=1) | The filesystem space available to non-root users (in bytes) |
| [rate(node\_network\_receive\_bytes\_total[1m])](http://localhost:9090/graph?g0.range_input=1h&g0.expr=rate(node_network_receive_bytes_total%5B1m%5D)&g0.tab=1) | The average network traffic received, per second, over the last minute (in bytes) |

1. download prometheus

2. extract tar file

in prometheus.yml

=================

# my global config

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# Alertmanager configuration

alerting:

alertmanagers:

- static\_configs:

- targets:

# - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.

rule\_files:

# - "first\_rules.yml"

# - "second\_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: "prometheus"

# metrics\_path defaults to '/metrics'

# scheme defaults to 'http'.

static\_configs:

- targets: ["localhost:9090"]

- job\_name: "springactuator"

metrics\_path: '/actuator/prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:8888']

3. ./prometheus

4. prometheus will be running in localhost:9090

5. download grafana

Standalone Linux Binaries(64 Bit)SHA256: fb50d6cee47b78ca5b21d3472ec76855df1d16970bd504a397ddedc69f24f068

wget https://dl.grafana.com/enterprise/release/grafana-enterprise-8.2.3.linux-amd64.tar.gz

tar -zxvf grafana-enterprise-8.2.3.linux-amd64.tar.gz

6. extract tar

7. to run grafana

cd bin

./grafana-server

it will run in localhost:3000

8. login -> admin/admin

skip to reset the password

9. run your spring boot APP

=========================================================================

REST API Web end points creation

pom.xml

<?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>3.2.11</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.stackroute</groupId>

<artifactId>grafana</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>grafana</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<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>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

spring.application.name=grafana

server.port=8888

management.endpoints.web.exposure.include=\*

management.endpoints.health.show-details=always

package com.stackroute.grafana.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/monitor")

public class HealthController {

@GetMapping("/health")

public String getHealth(){

// try{

// boolean condition=true;

// while(condition)

// {

// Runnable r=()->{

// while(true){}

// };

// new Thread(r).start();

// Thread.sleep(5000);

// }

// } catch (InterruptedException e) {

// throw new RuntimeException(e);

// }

return "welcome to stackroute";

}

}

10.

localhost:8888/monitor/health

localhost:8888/actuator

http://localhost:8888/actuator/prometheus

11. in prometheus web interface

verify

click option metric explorer near search bar execute button

select

go\_gc\_duration\_seconds\_count

see the graph view and table view

click Add panel

verify

do the same for

process\_cpu\_seconds\_total

do the same for

system\_cpu\_usage

12. in grafana web interface

add data source

Home-> connections-> data sources-> Add datasource

select the default Prometheus

name: prometheus2

connection url : http://localhost:9090

save and test button at bottom

we should get successfully queried the prometheus API

13. create a dashboard

Home-> dashboard -> new

Add visualization -> select the prometheus2

select metric

go\_gc\_duration\_seconds\_count

select filter

instance

value

localhost:9090

click runqueries

expland the view in panel

click save the dashboard

CPU usage metric

folder : dashboards

save

verify go the home

select the dashboard and view the panel

14. To visualize the predefined dashboard

create a new data source as premetheus 3

and set as default is very important

download: 12900\_rev3.json

13. + -> import dashboard

upload dashboard json file here

select the default prometheus which is marked as default

Alert rules for Prometheus:

download alert manager

https://github.com/shazforiot/Prometheusalertmanager

tar -zvzf alertmanager.. .tar.gz

cd alertmanager

cat alertmanager.yml

./alertmanager

localhost:9093

=================================

integrate with prometheus

cat prometheus.yml

alerting:

alertmaangers:

-static\_configs:

- targets:

- alertmanager:9093

==================================

prometheus.yml

rule\_files:

- "alert.rules.yml"

alerting:

alertmanagers:

- static\_configs:

- targets:

- 'localhost:9093'

=================================

./prometheus

status -> rules

status-> configuration

Alerts tab:

do one node exporter down

groups:

- name: alert.rules

rules:

- alert: InstanceDown

expr: up == 0

for: 1m

labels:

severity: "critical"

annotations:

summary: "Endpoint {{ $labels.instance }} down"

description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 1 minutes."

- alert: HostOutOfMemory

expr: node\_memory\_MemAvailable / node\_memory\_MemTotal \* 100 < 25

for: 5m

labels:

severity: warning

annotations:

summary: "Host out of memory (instance {{ $labels.instance }})"

description: "Node memory is filling up (< 25% left)\n VALUE = {{ $value }}\n LABELS: {{ $labels }}"

- alert: HostOutOfDiskSpace

expr: (node\_filesystem\_avail{mountpoint="/"} \* 100) / node\_filesystem\_size{mountpoint="/"} < 50

for: 1s

labels:

severity: warning

annotations:

summary: "Host out of disk space (instance {{ $labels.instance }})"

description: "Disk is almost full (< 50% left)\n VALUE = {{ $value }}\n LABELS: {{ $labels }}"

- alert: HostHighCpuLoad

expr: (sum by (instance) (irate(node\_cpu{job="node\_exporter\_metrics",mode="idle"}[5m]))) > 80

for: 5m

labels:

severity: warning

annotations:

summary: "Host high CPU load (instance {{ $labels.instance }})"

description: "CPU load is > 80%\n VALUE = {{ $value }}\n LABELS: {{ $labels }}"

In AlertManager.yml for email notification:

global:

resolve\_timeout: 1m

route:

receiver: 'email-notifications'

receivers:

- name: 'email-notifications'

email\_configs:

- to:

from:

smarthost: smtp.gmail.com:587

auth\_username:

auth\_identity:

auth\_password:

send\_resolved: true

**Splunk:**

$ wget https://download.splunk.com/products/splunk/releases/8.2.6/linux/splunk-8.2.6-a6fe1ee8894b-Linux-x86\_64.tgz

$ ls

$ tar -zxvf splunk-8.2.6-a6fe1ee8894b-Linux-x86\_64.tgz

$ clear

$ ls

$ cd bin

$ ls

$ ./splunk start --accept-license

$ administrator username: admin

password: saravanan123

======================================================================

1. cd splunk

2. cd bin

3. ./splunk start

The Splunk web interface is at http://ip-172-31-3-159:8000

Tutorial Demo:

1. source="tutorialdata.zip:\*" All time

2. source="tutorialdata.zip:./www1/access.log" All time

3. buttercupgames All time

4. "categoryid=sports" Date Range

since 2024

5. buttercupgames (error OR fail\* OR severe) All time

6. sourcetype=access\_\* All time

7. specify additional selected fields

To add fields to the selected Fields list, click All Fields at the top of Fields side bar.

add action, caregoryId, productId and close it

click action (gives field summary)

8. i column > expand this

to expand event information

9. sourcetype=access\_\* status=200 action=purchase All time

10. sourcetype=access\_\* status!=200 action=purchase

11. (error OR fail\* OR severe) OR (status=404 OR status=500 OR status=503)

12. sourcetype=access\_\* status=200 action=purchase categoryId=simulation

13. Locate the unique categoryId values by clicking on the categoryId field in the Selected Fields list.

Click on a categoryId name, such as ACCESSORIES. The categoryId is added to your search and the

search is automatically run again.

14. sourcetype=access\_\* status=200 action=purchase categoryId=simulation

15.

sourcetype=access\_\* status=200 action=purchase |

16. sourcetype=access\_\* status=200 action=purchase | top categoryId

The top command is a transforming command. Transforming commands organize the search results into a table. Use

transforming commands to generate results that you can use to create visualizations such as column, line, area, and pie

charts.

Because transforming commands return your search results in a table format, the results appear on the Statistics tab

17. sourcetype=access\_\* status=200 action=purchase | top categoryId

1. Click the Visualization tab.

By default, the Visualization tab opens with a Column chart.

2. Click Column Chart to open the visualization type selector.

select pie chart

move the mouse pointer to pie

18. click on top of pie chart and click categoryId: strategy

19. sourcetype=access\_\* status=200 action=purchase | top limit=1 clientip

20. sourcetype=access\_\* status=200 action=purchase clientip=87.194.216.51 | stats count,

distinct\_count(productId), values(productId) by clientip

21. sourcetype=access\_\* status=200 action=purchase | top limit=1 clientip | table clientip

22. sourcetype=access\_\* status=200 action=purchase [search sourcetype=access\_\* status=200

action=purchase | top limit=1 clientip | table clientip] | stats count, distinct\_count(productId),

values(productId) by clientip

23. sourcetype=access\_\* status=200 action=purchase [search sourcetype=access\_\* status=200

action=purchase | top limit=1 clientip | table clientip] | stats count AS "Total Purchased",

distinct\_count(productId) AS "Total Products", values(productId) AS "Product IDs" by clientip |

rename clientip AS "VIP Customer"

24. sourcetype=access\_\* status=200 action=purchase [search sourcetype=access\_\* status=200

action=purchase | top limit=1 clientip | table clientip] | stats count AS "Total Purchased",

dc(productId) AS "Total Products", values(productName) AS "Product Names" BY clientip | rename

clientip AS "VIP Customer"

save as -> report

Title: VIP customer

Description: Butercup Games most frequent shopper

Time Ranger picker : Yes

save

Contineu Editing

Add to Dashboard

view

click view

=====================================================================

Demo2:

settings -> data inputs-> local input-> http event collector go and verify

port number : 8088

copy token value: 507bd728-4c58-4c25-aac1-dac2e8f22cfb

copy source type: log4j

copy index : october

copy source: http-event-logs

========================================================================

splunk web interface

=======================

settings:

data inputs:

Http event collector->

global settings

all tokens enabled

default source type -> \_json

default index-> main

default output group -> one

http port number: 8088

save

==========================================================

new token

name: order-service-logs

source name override: http-event-logs

next-> source type

new -> select -> log4j

go down

create a new index

index name: order\_api\_dev

in that enable reduction and reduction days as 5

save

select new indexname to selected items

review and click submit

================================================================

Note: While creating index,

index name as saravanan, index data type is events, data integrity check is enabled, max size is 500 GB, Max size of Hot is auto GB, Tsidx Retention policy is enabled

reduction, reduce Tsidx file older is 10 hrs

=================================================================

Note : While creating data inputs -> http event collector ->

In global settings: All tokens - enabled, default source type is json, default index is default and port number is 8088

================================================================

to verify the output:

Apps-> search and reporting:

New search:

index="october"

================================================================

spring boot App

===============

pom.xml

<?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>2.7.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.javatechie</groupId>

<artifactId>splunk-demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>splunk-demo</name>

<description>Spring Boot Splunk integration</description>

<properties>

<java.version>1.8</java.version>

</properties>

<repositories>

<repository>

<id>splunk-artifactory</id>

<name>Splunk Releases</name>

<url>https://splunk.artifactoryonline.com/artifactory/ext-releases-local</url>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<artifactId>spring-boot-starter-logging</artifactId>

<groupId>org.springframework.boot</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>

<dependency>

<groupId>com.splunk.logging</groupId>

<artifactId>splunk-library-javalogging</artifactId>

<version>1.8.0</version>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</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>

<configuration>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

</build>

</project>

application.yml

server:

port: 9090

=========================================================================

log4j2-spring.xml

<?xml version="1.0" encoding="UTF-8"?>

<Configuration>

<Appenders>

<Console name="console" target="SYSTEM\_OUT">

<PatternLayout

pattern="%style{%d{ISO8601}} %highlight{%-5level }[%style{%t}{bright,blue}] %style{%C{10}}{bright,yellow}: %msg%n%throwable" />

</Console>

<SplunkHttp

name="splunkhttp"

url="http://localhost:8088"

token="507bd728-4c58-4c25-aac1-dac2e8f22cfb"

host="localhost"

index="october"

type="raw"

source="http-event-logs"

sourcetype="log4j"

messageFormat="text"

disableCertificateValidation="true">

<PatternLayout pattern="%m" />

</SplunkHttp>

</Appenders>

<Loggers>

<!-- LOG everything at INFO level -->

<Root level="info">

<AppenderRef ref="console" />

<AppenderRef ref="splunkhttp" />

</Root>

</Loggers>

</Configuration>

package com.javatechie.dto;

import lombok.AllArgsConstructor;

import lombok.Builder;

import lombok.Data;

import lombok.NoArgsConstructor;

import java.util.Date;

@AllArgsConstructor

@NoArgsConstructor

@Builder

@Data

public class Order {

private int id;

private String name;

private int qty;

private double price;

private String transactionId;

private Date orderPlacedDate;

}

package com.javatechie.service;

import com.javatechie.dto.Order;

import com.javatechie.util.Mapper;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

import java.util.UUID;

@Service

public class OrderService {

Logger logger= LogManager.getLogger(OrderService.class);

private List<Order> orderList = new ArrayList<>();

public Order addOrder(Order order) {

logger.info("OrderService:addOrder execution started..");

logger.info("OrderService:addOrder request payload {} ", Mapper.mapToJsonString(order));

order.setOrderPlacedDate(new Date());

order.setTransactionId(UUID.randomUUID().toString());

orderList.add(order);

logger.info("OrderService:addOrder response {} ", Mapper.mapToJsonString(order));

logger.info("OrderService:addOrder execution ended..");

return order;

}

public List<Order> getOrders() {

logger.info("OrderService:getOrders execution started..");

List<Order> list = null;

list = orderList;

logger.info("OrderService:getOrders response {} ", Mapper.mapToJsonString(orderList));

logger.info("OrderService:getOrders execution ended..");

return list;

}

public Order getOrder(int id) {

logger.info("OrderService:getOrder execution started..");

Order order = orderList.stream()

.filter(ord -> ord.getId() == id)

.findAny().orElseThrow(() -> new RuntimeException("Order not found with id : " + id));

logger.info("OrderService:getOrder execution ended..");

return order;

}

}

=========================================================================

package com.javatechie.util;

import com.fasterxml.jackson.core.JsonProcessingException;

import com.fasterxml.jackson.databind.ObjectMapper;

public class Mapper {

public static String mapToJsonString(Object object) {

try {

return new ObjectMapper().writeValueAsString(object);

} catch (JsonProcessingException e) {

e.printStackTrace();

}

return null;

}

}

=========================================================================

package com.javatechie;

import com.javatechie.dto.Order;

import com.javatechie.service.OrderService;

import com.javatechie.util.Mapper;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@SpringBootApplication

@RestController

@RequestMapping("/orders")

public class SplunkDemoApplication {

Logger logger = LogManager.getLogger(SplunkDemoApplication.class);

@Autowired

private OrderService service;

@PostMapping

public Order placeOrder(@RequestBody Order order) {

logger.info("OrderController:placeOrder persist order request {}", Mapper.mapToJsonString(order));

Order addOrder = service.addOrder(order);

logger.info("OrderController:placeOrder response from service {}", Mapper.mapToJsonString(addOrder));

return addOrder;

}

@GetMapping

public List<Order> getOrders() {

List<Order> orders = service.getOrders();

logger.info("OrderController:getOrders response from service {}", Mapper.mapToJsonString(orders));

return orders;

}

@GetMapping("/{id}")

public Order getOrder(@PathVariable int id) {

logger.info("OrderController:getOrder fetch order by id {}", id);

Order order = service.getOrder(id);

logger.info("OrderController:getOrder fetch order response {}", Mapper.mapToJsonString(order));

return order;

}

public static void main(String[] args) {

SpringApplication.run(SplunkDemoApplication.class, args);

}

}

==========================================================

In postman :

Post: localhost:9090/orders

Body -> raw -> json

{

"id":21,

"name": "soaps",

"qty": 100,

"price": 4500

}

submit

ensure 200 ok

verify logs in console also.

===========================================================

**ELK:**

Setting up the ELK (Elasticsearch, Logstash, Kibana) stack for centralized logging involves multiple steps. Below are step-by-step instructions to help you set up ELK stack on a local machine for development purposes.

Step 1: Install Elasticsearch

===============================

1. Download and Extract Elasticsearch:

- Visit the Elasticsearch download page: https://www.elastic.co/downloads/elasticsearch

- Download the appropriate version for your operating system.

- Extract the downloaded archive to your preferred location.

2. Run Elasticsearch:

- Navigate to the Elasticsearch directory.

- Run Elasticsearch using the following command:

bash

bin/elasticsearch

Note: first time loading, all the log information to get stored like password, token etc..

- Elasticsearch should start, and you can access the server at `http://localhost:9200` in your browser.

Step 2: Install Kibana

======================

1. Download and Extract Kibana:

- Visit the Kibana download page: https://www.elastic.co/downloads/kibana

- Download the appropriate version for your operating system.

- Extract the downloaded archive to your preferred location.

2. Configure Kibana:

- Open the `config/kibana.yml` file.

- Uncomment and set the `elasticsearch.hosts` property to point to your Elasticsearch instance:

yaml

elasticsearch.hosts: ["http://localhost:9200"]

3. Run Kibana:

- Navigate to the Kibana directory.

- Run Kibana using the following command:

bash

bin/kibana

- Access Kibana at `http://localhost:5601` in your browser.

Step 3: Install Logstash

========================

1. Download and Extract Logstash:

- Visit the Logstash download page: https://www.elastic.co/downloads/logstash

- Download the appropriate version for your operating system.

- Extract the downloaded archive to your preferred location.

2. Create Logstash Configuration File:

- Create a Logstash configuration file (e.g., `logstash.conf`) with the following content:

conf

input {

file {

path => "/path/to/your/application/logs/application.log"

start\_position => "beginning"

}

}

output {

elasticsearch {

hosts => ["localhost:9200"]

index => "spring-boot-logs"

}

stdout {}

}

- Update the `path` to match the path of your application logs.

3. Run Logstash:

- Navigate to the Logstash directory.

- Run Logstash using the following command:

bash

bin/logstash -f logstash.conf

Step 4: Test the Setup

1. Run your Spring Boot application to generate logs.

2. Open Kibana at `http://localhost:5601` in your browser.

3. Set up an index pattern in Kibana:

- Go to the "Management" section in Kibana.

- Click on "Index Patterns" and create an index pattern for "spring-boot-logs."

4. Explore your logs in the "Discover" section of Kibana.

Note: Keep in mind that for production use, we should follow best practices for security and scalability.

**Tracing:**

===================================================

sudo chmod 666 /var/run/docker.sock

sudo service docker start

docker ps

docker images

================================================================================================

pom.xml

========

<?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>3.2.11</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.stackroute</groupId>

<artifactId>teledemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>teledemo</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<!-- <dependency>-->

<!-- <groupId>org.slf4j</groupId>-->

<!-- <artifactId>slf4j-api</artifactId>-->

<!-- <version>1.7.30</version> &lt;!&ndash; Check for the latest version &ndash;&gt;-->

<!-- </dependency>-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- <dependency>-->

<!-- <groupId>io.jaegertracing</groupId>-->

<!-- <artifactId>jaeger-client</artifactId>-->

<!-- <version>1.8.0</version> &lt;!&ndash; Check for the latest version &ndash;&gt;-->

<!-- </dependency>-->

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-tracing-bridge-otel</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-exporter-otlp</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</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>

<configuration>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

</build>

</project>

application.yaml

================

server:

port: 8090

spring:

application:

name: app2

tracing:

url: http://localhost:4318/v1/traces

management:

tracing:

sampling:

probability: 1.0

logging:

pattern:

level: "%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]"

#spring.application.name=teledemo

#jaeger.endpoint=http://jaeger:14268/api/traces

#jaeger.service-name=teledemo # Ensure this matches the app name

##spring.sleuth.sampler.probability=1.0 # Ensure all traces are sampled

#management.endpoints.web.exposure.include=\*

#management.endpoint.health.show-details=always

#server.port=8085

#logging.file=/home/ubuntu/Desktop/logs/elk-stack.log

#logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level [%thread] [%X{traceId}] [%X{spanId}] %logger{36} - %msg%n

package com.stackroute.teledemo.config;

import io.opentelemetry.exporter.otlp.http.trace.OtlpHttpSpanExporter;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.web.client.RestTemplateBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class AppConfig {

@Bean

OtlpHttpSpanExporter otlpHttpSpanExporter(@Value("${tracing.url}")

String url){

return OtlpHttpSpanExporter.builder()

.setEndpoint(url)

.build();

}

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

package com.stackroute.teledemo.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import org.springframework.beans.factory.annotation.Value;

@RestController

@RequestMapping("/service")

public class HomeController {

private static Logger logger = LoggerFactory.getLogger(HomeController.class);

private RestTemplate restTemplate;

public HomeController(RestTemplate restTemplate)

{

this.restTemplate=restTemplate;

}

@Value("${spring.application.name}")

private String applicationName; // Set your application name

@GetMapping("/path1")

public ResponseEntity<String> path1() {

logger.info("Incoming request at {} for request/path1");

String response = restTemplate.getForObject("http://localhost:8090/service/path2", String.class);

return ResponseEntity.ok("Response from /path1: " + response);

}

@GetMapping("/path2")

public ResponseEntity<String> path2() {

logger.info("Incoming request at path1");

// String response = restTemplate.getForObject("http://localhost:8090/service/path2", String.class);

return ResponseEntity.ok("Response from path2: ");

}

}

package com.stackroute.teledemo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RestController;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

@SpringBootApplication

@RestController

public class TeledemoApplication {

public static void main(String[] args) {

SpringApplication.run(TeledemoApplication.class, args);

}

}

==================================================================

docker-compose up --build

localhost:16686/search

first run:

spring.application.name=app1

server.port=8081

second run:

spring.application.name=app2

server.port=8090

postman or browser:

http://localhost:8081/service/path1

response from /path1 + response from /path2

verify the service logs in console

both terminal

verify trace id, span id in both terminal

localhost:16686/search

app1

app2

Demo2:

**Getting Started by Example**

Get telemetry for your app in less than 5 minutes!

This page will show you how to get started with OpenTelemetry in Java.

You will learn how you can instrument a simple Java application automatically, in such a way that [traces](https://opentelemetry.io/docs/concepts/signals/traces/), [metrics](https://opentelemetry.io/docs/concepts/signals/metrics/), and [logs](https://opentelemetry.io/docs/concepts/signals/logs/) are emitted to the console.

Prerequisites

Ensure that you have the following installed locally:

* Java JDK 17+, due to the use of Spring Boot 3; [Java 8+ otherwise](https://github.com/open-telemetry/opentelemetry-java/blob/main/VERSIONING.md#language-version-compatibility)
* [Gradle](https://gradle.org/)

Example Application

The following example uses a basic [Spring Boot](https://spring.io/guides/gs/spring-boot/) application. You can use another web framework, such as Apache Wicket or Play. For a complete list of libraries and supported frameworks, consult the [registry](https://opentelemetry.io/ecosystem/registry/?component=instrumentation&language=java).

For more elaborate examples, see [examples](https://opentelemetry.io/docs/languages/java/examples/).

Dependencies

To begin, set up an environment in a new directory called java-simple. Within that directory, create a file called build.gradle.kts with the following content:

plugins **{**

id**(**"java"**)**

id**(**"org.springframework.boot"**)** version "3.0.6"

id**(**"io.spring.dependency-management"**)** version "1.1.0"

**}**

sourceSets **{**

main **{**

java**.**setSrcDirs**(**setOf**(**"."**))**

**}**

**}**

repositories **{**

mavenCentral**()**

**}**

dependencies **{**

implementation**(**"org.springframework.boot:spring-boot-starter-web"**)**

**}**

Create and launch an HTTP Server

In that same folder, create a file called DiceApplication.java and add the following code to the file:

**package** otel**;**

**import** org.springframework.boot.Banner**;**

**import** org.springframework.boot.SpringApplication**;**

**import** org.springframework.boot.autoconfigure.SpringBootApplication**;**

**@SpringBootApplication**

**public** **class** DiceApplication **{**

**public** **static** **void** main**(**String**[]** args**)** **{**

SpringApplication app **=** **new** SpringApplication**(**DiceApplication**.**class**);**

app**.**setBannerMode**(**Banner**.**Mode**.**OFF**);**

app**.**run**(**args**);**

**}**

**}**

Create another file called RollController.java and add the following code to the file:

**package** otel**;**

**import** java.util.Optional**;**

**import** java.util.concurrent.ThreadLocalRandom**;**

**import** org.slf4j.Logger**;**

**import** org.slf4j.LoggerFactory**;**

**import** org.springframework.web.bind.annotation.GetMapping**;**

**import** org.springframework.web.bind.annotation.RequestParam**;**

**import** org.springframework.web.bind.annotation.RestController**;**

**@RestController**

**public** **class** RollController **{**

**private** **static** **final** Logger logger **=** LoggerFactory**.**getLogger**(**RollController**.**class**);**

**@GetMapping(**"/rolldice"**)**

**public** String index**(@RequestParam(**"player"**)** Optional**<**String**>** player**)** **{**

**int** result **=** **this.**getRandomNumber**(**1**,** 6**);**

**if** **(**player**.**isPresent**())** **{**

logger**.**info**(**"{} is rolling the dice: {}"**,** player**.**get**(),** result**);**

**}** **else** **{**

logger**.**info**(**"Anonymous player is rolling the dice: {}"**,** result**);**

**}**

**return** Integer**.**toString**(**result**);**

**}**

**public** **int** getRandomNumber**(int** min**,** **int** max**)** **{**

**return** ThreadLocalRandom**.**current**().**nextInt**(**min**,** max **+** 1**);**

**}**

**}**

Build and run the application with the following command, then open <http://localhost:8080/rolldice> in your web browser to ensure it is working.

gradle assemble

java -jar ./build/libs/java-simple.jar

Instrumentation

Next, you’ll use a [Java agent](https://opentelemetry.io/docs/zero-code/java/agent/) to automatically instrument the application at launch time. While you can [configure the Java agent](https://opentelemetry.io/docs/zero-code/java/agent/configuration/) in a number of ways, the steps below use environment variables.

1. Download [opentelemetry-javaagent.jar](https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/latest/download/opentelemetry-javaagent.jar) from [Releases](https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases) of the opentelemetry-java-instrumentation repository. The JAR file contains the agent and all automatic instrumentation packages:
2. *curl -L -O https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/latest/download/opentelemetry-javaagent.jar*

 Take note of the path to the JAR file.

1. Set and export variables that specify the Java agent JAR and a [console exporter](https://github.com/open-telemetry/opentelemetry-java/blob/main/sdk-extensions/autoconfigure/README.md#logging-exporter), using a notation suitable for your shell/terminal environment — we illustrate a notation for bash-like shells:
2. export JAVA\_TOOL\_OPTIONS**=**"-javaagent:PATH/TO/opentelemetry-javaagent.jar" \
3. OTEL\_TRACES\_EXPORTER**=**logging \
4. OTEL\_METRICS\_EXPORTER**=**logging \
5. OTEL\_LOGS\_EXPORTER**=**logging \
6. OTEL\_METRIC\_EXPORT\_INTERVAL**=15000**

Important

* + Replace PATH/TO above, with your path to the JAR.
  + Set OTEL\_METRIC\_EXPORT\_INTERVAL to a value well below the default, as we illustrate above, **only during testing** to help you more quickly ensure that metrics are properly generated.

1. Run your **application** once again:
2. $ java -jar ./build/libs/java-simple.jar
3. *...*

Note the output from the otel.javaagent.

1. From *another* terminal, send a request using curl:
2. curl localhost:8080/rolldice
3. Stop the server process.

At step 4, you should have seen trace & log output from the server and client that looks something like this (trace output is line-wrapped for convenience):

**[**otel.javaagent 2023-04-24 17:33:54:567 +0200**]** **[**http-nio-8080-exec-1**]** INFO

io.opentelemetry.exporter.logging.LoggingSpanExporter - 'RollController.index' :

70c2f04ec863a956e9af975ba0d983ee 7fd145f5cda13625 INTERNAL **[**tracer:

io.opentelemetry.spring-webmvc-6.0:1.25.0-alpha**]** AttributesMap**{**data**=**

**{**thread.id**=**39, thread.name**=**http-nio-8080-exec-1**}**, capacity**=**128,

totalAddedValues**=**2**}**

**[**otel.javaagent 2023-04-24 17:33:54:568 +0200**]** **[**http-nio-8080-exec-1**]** INFO

io.opentelemetry.exporter.logging.LoggingSpanExporter - 'GET /rolldice' :

70c2f04ec863a956e9af975ba0d983ee 647ad186ad53eccf SERVER **[**tracer:

io.opentelemetry.tomcat-10.0:1.25.0-alpha**]** AttributesMap**{**

data**={**user\_agent.original**=**curl/7.87.0, net.host.name**=**localhost,

net.transport**=**ip\_tcp, http.target**=**/rolldice, net.sock.peer.addr**=**127.0.0.1,

thread.name**=**http-nio-8080-exec-1, net.sock.peer.port**=**53422,

http.route**=**/rolldice, net.sock.host.addr**=**127.0.0.1, thread.id**=**39,

net.protocol.name**=**http, http.status\_code**=**200, http.scheme**=**http,

net.protocol.version**=**1.1, http.response\_content\_length**=**1,

net.host.port**=**8080, http.method**=**GET**}**, capacity**=**128, totalAddedValues**=**17**}**

At step 5, when stopping the server, you should see an output of all the metrics collected (metrics output is line-wrapped and shortened for convenience):

**[**otel.javaagent 2023-04-24 17:34:25:347 +0200**]** **[**PeriodicMetricReader-1**]** INFO

io.opentelemetry.exporter.logging.LoggingMetricExporter - Received a collection

of **19** metrics **for** export.

**[**otel.javaagent 2023-04-24 17:34:25:347 +0200**]** **[**PeriodicMetricReader-1**]** INFO

io.opentelemetry.exporter.logging.LoggingMetricExporter - metric:

ImmutableMetricData**{**resource**=**Resource**{**schemaUrl**=**

https://opentelemetry.io/schemas/1.19.0, attributes**={**host.arch**=**"aarch64",

host.name**=**"OPENTELEMETRY", os.description**=**"Mac OS X 13.3.1", os.type**=**"darwin",

process.command\_args**=[**/bin/java, -jar, java-simple.jar**]**,

process.executable.path**=**"/bin/java", process.pid**=**64497,

process.runtime.description**=**"Homebrew OpenJDK 64-Bit Server VM 20",

process.runtime.name**=**"OpenJDK Runtime Environment",

process.runtime.version**=**"20", service.name**=**"java-simple",

telemetry.auto.version**=**"1.25.0", telemetry.sdk.language**=**"java",

telemetry.sdk.name**=**"opentelemetry", telemetry.sdk.version**=**"1.25.0"**}}**,

instrumentationScopeInfo**=**InstrumentationScopeInfo**{**name**=**io.opentelemetry.runtime-metrics,

version**=**1.25.0, schemaUrl**=**null, attributes**={}}**,

name**=**process.runtime.jvm.buffer.limit, description**=**Total capacity of the buffers

in this pool, unit**=**By, type**=**LONG\_SUM, data**=**ImmutableSumData**{**points**=**

**[**ImmutableLongPointData**{**startEpochNanos**=**1682350405319221000,

epochNanos**=**1682350465326752000, attributes**=**

**{**pool**=**"mapped - 'non-volatile memory'"**}**, value**=**0, exemplars**=[]}**,

ImmutableLongPointData**{**startEpochNanos**=**1682350405319221000,

epochNanos**=**1682350465326752000, attributes**={**pool**=**"mapped"**}**,

value**=**0, exemplars**=[]}**,

ImmutableLongPointData**{**startEpochNanos**=**1682350405319221000,

epochNanos**=**1682350465326752000, attributes**={**pool**=**"direct"**}**,

value**=**8192, exemplars**=[]}]**, monotonic**=**false, aggregationTemporality**=**CUMULATIVE**}}**

...