**Prometheus**

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. It is now a standalone open-source project maintained independently of any company. Prometheus is part of the Cloud Native Computing Foundation (CNCF), which is a part of the Linux Foundation. Here's an overview of Prometheus:

**Key Features:**

* Multi-dimensional Data Model: Prometheus stores time-series data with metrics identified by a metric name and a set of key-value pairs.
* PromQL (Prometheus Query Language): PromQL is a powerful query language that allows users to select and aggregate time-series data for analysis, graphing, and alerting.
* Pull-based Metrics Collection: Prometheus scrapes metrics data from instrumented targets, typically using HTTP over a pull mechanism. This allows it to be very scalable and to support a variety of monitoring needs.
* Service Discovery: Prometheus can automatically discover and monitor targets in dynamic environments like Kubernetes, AWS, or other cloud-based systems.
* Time-series Database: Prometheus has its own time-series database optimized for high write throughput and efficient querying of time-series data.
* Data Exporters and Integrations: There are many third-party exporters available to expose metrics from various systems and applications. Additionally, Prometheus can integrate with Grafana for data visualization.

**Use Cases:**

* Monitoring Microservices: Prometheus is well-suited for monitoring microservices architectures, providing insights into the performance and health of individual services as well as the overall system.
* Infrastructure Monitoring: It can monitor servers, databases, and other infrastructure components to ensure they are operating as expected.
* Container Orchestration Platforms: Prometheus integrates seamlessly with container orchestration platforms like Kubernetes for monitoring the health and performance of containers and pods.
* Application Performance Monitoring (APM): While Prometheus primarily focuses on infrastructure monitoring, it can also be used for basic application performance monitoring by exposing relevant metrics through custom exporters.

**Demo Project was done to illustrate integration of Prometheus.**

**Dependencies used for the project:**

<dependency>

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

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

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

<scope>runtime</scope>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.2.0</version>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

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

<scope>runtime</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.6.9</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-boot-starter</artifactId>

<version>3.0.0</version> <!-- Use the latest version -->

</dependency>

<dependency>

<groupId>com.fasterxml</groupId>

<artifactId>classmate</artifactId>

<version>1.5.1</version> <!-- Use the latest version -->

</dependency>

<dependency>

<groupId>com.github.ulisesbocchio</groupId>

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

<version>3.0.5</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

</dependency>

**EmpController Class:**

@RestController

@RequestMapping (path = "/employees")

@OpenAPI30

public class EmpController {

@Autowired

private EmpService empService;

@PostMapping("/save")

@Timed (value = "requests.count.save")

public Employee createEmployee(@RequestBody Employee employee) {

return empService.createEmployee(employee);

}

@GetMapping("/getAll")

@Timed (value = "requests.count.getAll")

public List<Employee> getAllEmployees() {

return empService.getAllEmployees();

}

@GetMapping("/get/{empId}")

@Timed (value = "requests.count.getById")

public Optional<Employee> getEmployeeById(@PathVariable Integer empId) {

return empService.getEmployeeById(empId);

}

@PutMapping("/update")

@Timed (value = "requests.count.update")

public Employee updateEmployee(@RequestBody Employee updatedEmployee) throws NotFoundException {

return empService.updatedEmployee(updatedEmployee);

}

@DeleteMapping("/delete/{empId}")

@Timed (value = "requests.count.delete")

public void deleteEmployee(@PathVariable Integer empId) throws NotFoundException {

empService.deleteEmployee(empId);

}

@GetMapping("/getbyFirstName/{firstName}")

public Employee getbyFirstName(@PathVariable String firstName) {

return empService.getEmployee(firstName);

}

@Value ("${my. param}")

private String secretparam;

@GetMapping("/testEncrypt")

public void test2(@RequestHeader (required = false) String string) {

StandardPBEStringEncryptor se = new StandardPBEStringEncryptor();

se.setPassword("superkey");

String str = se.encrypt(string);

System.out.println(str);

}

@GetMapping("/testDecrypt")

public String test3()

{

return secretparam;

}

@GetMapping("/hello")

@Timed (value = "requests.count.hello")

public String sayHello()

{

return "Hello World";

}

**Prometheus Steps:**

* Add Dependency- Prometheus and Actuator
* Configuration in application properties file

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

* Configuration in promethus yaml file

- job\_name: "EmpDataService"

metrics\_path: "/actuator/prometheus"

scrape\_interval: 5s

static\_configs:

- targets: ["localhost:8089"]

* Click on Prometheus.exe file.
* Verify by checking with gc in the Prometheus Dashboard
* Annotate with @Timed along with the variable name for each method in the controller class.
* Add the below Bean method in the main class.

@Bean

TimedAspect timedAspect(MeterRegistry registry) {

return new TimedAspect(registry);

}

* Hit the Apis for the smooth graph.
* Hit the <http://localhost:8089/actuator/prometheus> endpoint and Press Ctrl+F and search for the variable which is used with @Timed Annotation
* With the same variable name search the Prometheus Dashboard, select the option and click on execute.
* Graph will be obtained.

A screenshot of a computer

Description automatically generated

**Requests for the Prometheus Dashboard**

* requests\_count\_getById\_seconds\_max:

The term "requests\_count\_getById\_seconds\_max" seems to be a metric name, possibly related to monitoring or performance tracking. It appears to follow the convention used by Prometheus, a popular monitoring and alerting toolkit.

* requests\_count\_getById\_seconds\_count:

Gives the total number of hits at the particular time and when was the last time it was hit.

* requests\_count\_getById\_seconds\_sum:

"requests\_count\_getById": This part suggests that it's counting the number of requests made to retrieve data by ID.

"\_seconds\_sum": This part typically indicates that it's measuring the sum of response times for these requests, likely in seconds.

* Gc and mem for garbage collection and memory used.

**Grafana**

Grafana is an open-source analytics and visualization platform designed for monitoring and observability. It allows users to query, visualize, alert on, and understand metrics no matter where they are stored. Originally built for Graphite metrics storage and visualization, Grafana has evolved to support various data sources, including but not limited to Prometheus, InfluxDB, Elasticsearch, MySQL, PostgreSQL, and many others.

**Key Features:**

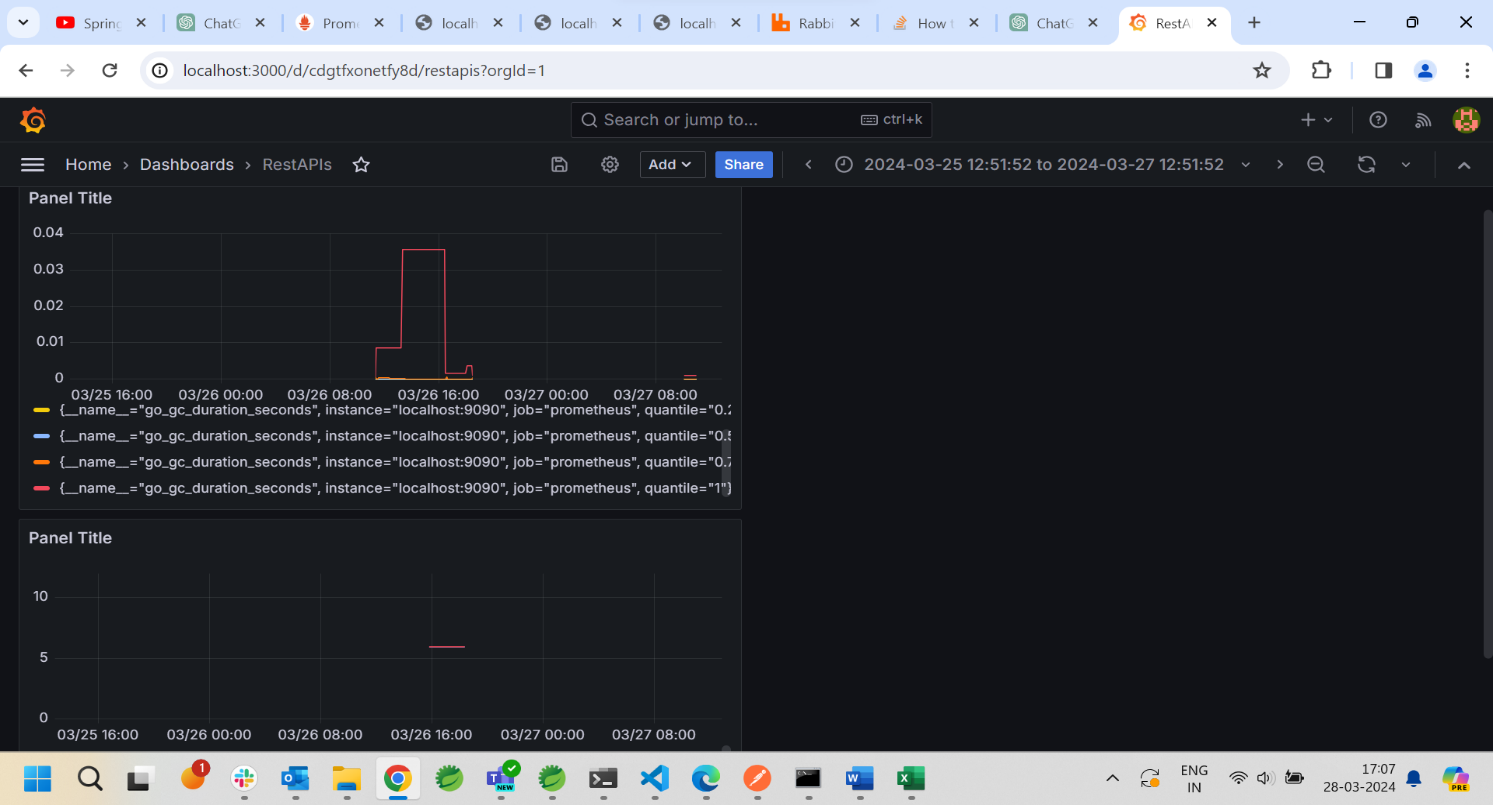
* Data Source Integration: Grafana supports a wide range of data sources, enabling users to connect to different databases, monitoring systems, time series databases, and logging platforms. This flexibility allows users to consolidate data from disparate sources into a single dashboard for unified monitoring and analysis.
* Visualization: Grafana offers a rich set of visualization options including graphs, charts, tables, gauges, histograms, and heatmaps. Users can customize the appearance of these visualizations, such as color schemes, line styles, and annotations, to convey data insights effectively.
* Dashboard Creation: Users can create interactive dashboards by arranging multiple panels to display different metrics or data sets. Dashboards can be organized into folders, shared with teams, and exported/imported for collaboration or backup purposes.
* Templating: Grafana supports templating, allowing users to create dynamic dashboards that adjust based on variables such as time ranges, hosts, data sources, or other parameters. This feature is useful for creating reusable dashboards across different environments or for exploring data interactively.
* Alerting and Notifications: Grafana provides built-in alerting functionality, allowing users to define alert rules based on threshold conditions or complex queries. When an alert is triggered, Grafana can send notifications via various channels including email, Slack, PagerDuty, and others.
* Community and Ecosystem: Grafana has a vibrant community of users, developers, and contributors who actively share dashboards, plugins, and knowledge through forums, meetups, and online resources. This ecosystem contributes to the platform's growth, adoption, and innovation.

**Use Cases:**

* Infrastructure Monitoring: Grafana is commonly used for monitoring server metrics, network performance, and resource utilization in IT infrastructure environments.
* Application Performance Monitoring (APM): Grafana can visualize metrics related to application performance, response times, error rates, and user interactions to identify performance bottlenecks and optimize application behavior.
* Business Intelligence (BI): Grafana is used for business analytics and reporting by integrating with databases, data warehouses, and BI platforms to create interactive dashboards for data-driven decision-making.
* IoT Monitoring: Grafana can aggregate and visualize sensor data from IoT devices, enabling real-time monitoring, anomaly detection, and predictive maintenance in industrial, smart city, and healthcare applications.

**Steps for Grafana**

* Go to Grafana source folder and click on the Grafana.exe file
* Go to connections.
* Add the Prometheus as the Data Source and give the localhost:9090 in the configuration.
* Select the option as Code.
* In Metric browser, give the variable name given in the Prometheus as it is
* Click on run queries.
* Add queries. (B will be created)
* Do the same steps.
* For the APM dashboard
* Click on “+” on top right corner.
* Import json file and attach the Json file.
* We will get the Detailed Dashboard



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated