**Code Monitoring Guidelines**

Objectives

There are several benefits of practicing code-level monitoring:

1. Tracking of overall application health
   1. Error/Success Rate
   2. Request/Response Throughput
   3. Request/Response Latency
2. Identifying potential logic bottlenecks
3. Identifying potential security vulnerabilities

What to Monitor?

There are four basic metrics that every application should measure/track:

1. Code execution count
2. Code execution error count
3. Code execution duration
4. Key business metrics

**Code Execution Count**

This metric is to track how many times a particular piece of code is being called/executed. The purpose of this metric is to provide a better understanding of the usage of production codes:

* what piece of code is frequently executed
* what piece of code is infrequently executed
* is the actual frequency of code execution in line with expectation?
* what are the code execution patterns, if there is any?

**Code Execution Error Count**

This metric is to track how many times a specific error is thrown in the application. The purpose of this metric helps to keep track the overall integrity of the codes deployed to production:

* what is the overall exception rate of the application (number of requests with exception/number of requests)
* is the overall exception rate within predefined expectation?
* what type of exception is being thrown
* known vs unknown
* internal vs external
* does the exception take place during a specific time/day - is there a pattern?

**Code Execution Duration**

This metric is to track the duration needed to execute a particular piece of code. The purpose of this metric is to help us monitor the efficiency of our production codes:

* what is the min, average, max and current execution duration for frequently executed code?
* what is the min, average, max and current execution duration for infrequently executed code?
* is there a big difference between the min, average and max execution duration?
* under what condition do min and max execution duration take place
* what are the potential bottlenecks in terms of execution duration
* is the actual execution duration in line with expectation?

**Key Business Metrics**

This metric identifies and tracks the high-level metrics that have a direct impact on the business. The purpose of this metric is to help us understand the code feature/functionality of the application in relation to the business. Key business metrics differ from application to application and they are tied to the domain of the specific application. Following table lists out the potential key business metrics for different microservices:

| **Service** | **Key Business Metrics** | **Application Metrics** |
| --- | --- | --- |
| User | * Successful user sign-up * Successful user login | * Rate of successful user sign-up (Number of successful user sign-up/Number of user sign-up attempts) * Rate of successful user login (Number of successful user login/Number of user login attempts) * Count of user sign-up errors * Count of user login errors * Duration of user sign up * Duration of user login |
| Policy | * Successful policy creation | * Rate of successful policy creation (Number of successful policy creation/Number of policy creation attempts) * Count of policy creation errors * Duration of policy creation |

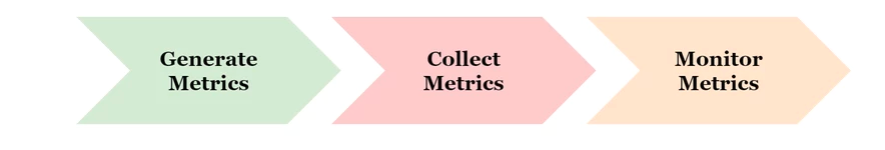
Where to Monitor?

Too much of a good thing is bad. As the code base grows larger, we have to be selective in terms of the number of metrics that we gather. The following is a guideline on where to implement code-level monitoring:

* New feature: all new features should be instrumented before they are deployed to production
* Codes related to key business metrics
* Database calls
* External calls
* 3rd party API
* other internal API
* Frequently accessed codes

How to Monitor?

The selected tool for code-level monitoring is [Prometheus](https://prometheus.io/docs/introduction/overview/): an open-source platform originally created by SoundCloud. On top of that, the visualization platform of choice is [Grafana](https://grafana.com/). The following is the general process of generating, collecting, monitoring, and visualizing code-level metrics:



**Generating Metrics**

* Non Spring Boot application
* Prometheus offers a wide range of client libraries for applications written in various languages. Use one of the Prometheus client libraries to register custom metrics within the application
* [Prometheus client libraries](https://prometheus.io/docs/instrumenting/clientlibs/)
* Spring Boot application
* We can take advantage of Spring Actuator to help us expose metrics within the Spring Boot application
* Using Prometheus Java client to track custom metrics is also possible

**Collecting Metrics**

* Pull versus Push method
* Prometheus supports both pull and push methods to collect application metrics
* The pull method should be utilized unless the target application is shortlived (e.g.: a cronjob)
* Cronjob and any other ephemeral applications can push the metrics to [Prometheus Pushgateway](https://prometheus.io/docs/instrumenting/pushing/) instead
* Pull versus Push reasoning
* The main reason pull method is preferred over the push method is that the pull method allows Prometheus to easily detect if a target application is down
  + When the target application is inaccessible/down,
    - Using the pull method: Prometheus will be unable to pull any metrics from the target application, therefore, concluding that the application is down
    - Using the push method: Prometheus will be unable to determine if the target application is down or simply does not have metrics to push over

**Monitoring Metrics**

* Grafana allows us to connect to various [data sources](https://grafana.com/plugins?type=datasource) including Prometheus
* Once Grafana is wired up to Prometheus, all metrics collected by Prometheus will be accessible in Grafana for the creation of various tracking/monitoring dashboard
* Different [alerting systems](http://docs.grafana.org/alerting/rules/) can also be set up for each dashboard on Grafana to trigger notification alert to the respective personnel if/when the target metric satisfies the set of predefined conditions
* **Example**: email application owner when the successful user login rate for the past 60 minutes drops below 70%

**Enabling Code Level Monitoring on Spring Boot Project with Spring Actuator**

The following details can also be found in the sample application: [Mock User API](https://bitbucket.org/ams-asia-regional/userapi/src/master/)

1. Add spring-boot-starter-actuator, micrometer-core and micrometer-registry-prometheus dependencies to the POM file
   1. **POM File**

|  |
| --- |
| <!-- Spring boot actuator to expose metrics endpoint -->  <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- Micormeter core dependency  -->  <dependency>      <groupId>io.micrometer</groupId>      <artifactId>micrometer-core</artifactId>  </dependency>  <!-- Micrometer Prometheus registry  -->  <dependency>      <groupId>io.micrometer</groupId>      <artifactId>micrometer-registry-prometheus</artifactId>  </dependency> |

1. Modify application.properties to enable Prometheus endpoint
   1. **application.properties**

|  |
| --- |
| # Metrics related configurations  management.endpoint.metrics.enabled=true  management.endpoint.prometheus.enabled=true  management.endpoints.web.exposure.include=health,info,prometheus,metrics  management.metrics.export.prometheus.enabled=true |

1. Implement a class to collect metrics
   1. **MetricsService**

|  |
| --- |
| public class MetricsService {      public synchronized static void incrementCounter(String counterName, String... tags){          Metrics.counter(counterName, tags).increment();      }        public synchronized static void incrementCounter(String counterName, double value, String... tags){          Metrics.counter(counterName, tags).increment(value);      }        public synchronized static void observeTime(String timerName, long duration, String... tags){          Metrics.timer(timerName, tags).record(duration, TimeUnit.MILLISECONDS);      }  } |

1. Pass PrometheusMeterRegistry to all services that require metrics tracking
   1. **Constructor**

|  |
| --- |
| public UserService(PrometheusMeterRegistry prometheusMeterRegistry) {          Metrics.addRegistry(prometheusMeterRegistry);  } |

1. Start tracking counter and timer at all point of interested within the code base

|  |
| --- |
| public List<User> getAllUsers() {          long start = System.currentTimeMillis();          List<User> users = userRepository.findAll();            // Track request duration for getAllUsers          MetricsService.observeTime(TIMER\_NAME, System.currentTimeMillis() - start, "method", "get.all");          // Track request count for getAllUsers          MetricsService.incrementCounter(COUNTER\_NAME, "method", "get.all");          return users;      }        public User getUserByUsername(String username) {          long start = System.currentTimeMillis();          User user = userRepository.findByUsername(username);            // Track request duration for getUserByUsername          MetricsService.observeTime(TIMER\_NAME, System.currentTimeMillis() - start, "method", "get.by.username");            if (user == null) {              // Track request error count for getUserByUsername              MetricsService.incrementCounter(ERROR\_COUNTER\_NAME, "method", "get.by.username", "error", "nonexistent.user", "validation.error", "na");          }          // Track request count for getUserByUsername          MetricsService.incrementCounter(COUNTER\_NAME, "method", "get.by.username");          return user;     } |

1. Verify that metrics are successfully collected through Spring Actuator Prometheus endpoint: http://BASE\_PATH/actuator/prometheus