**Best Practices For Application Deployment**

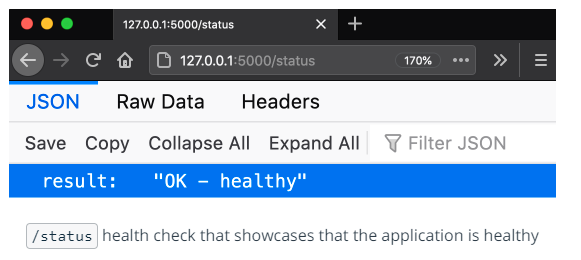
**Summary**

Using the knowledge acquired so far, you should be able to choose the most suitable architecture for an application, based on requirements, available resources, and involved trade-offs. The next stage consists of building the application. Regardless of the chosen architecture, a set of good development practices can be applied to improve the application lifecycle throughout the release and maintenance phases. Adopting these practices increases resiliency, lowers the time to recovery, and provides transparency of how a service handles incoming requests.

These practices are focused on health checks, metrics, logs, tracing, and resource consumption.

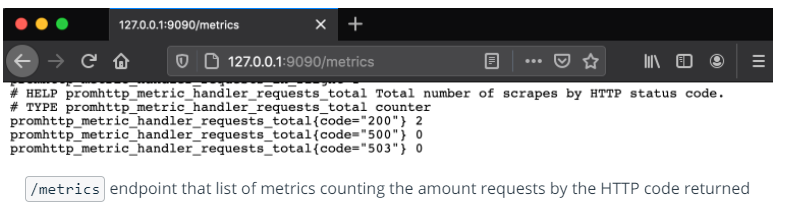
**Health Checks**

Health checks are implemented to showcase the status of an application. These checks report if an application is running and meets the expected behavior to serve incoming traffic. Usually, health checks are represented by an HTTP endpoint such as /healthz or /status. These endpoints return an HTTP response showcasing if the application is healthy or in an error state.



**Metrics**

Metrics are necessary to quantify the performance of the application. To fully understand how a service handles requests, it is mandatory to collect statistics on how the service operates. For example, the number of active users, handled requests, or the number of logins. Additionally, it is paramount to gather statistics on resources that the application requires to be fully operational. For example, the amount of CPU, memory, and network throughput. Usually, the collection of metrics are returned via an HTTP endpoint such as /metrics, which contains the internal metrics such as the number of active users, consumed CPU, network throughput, etc.



**Logs**

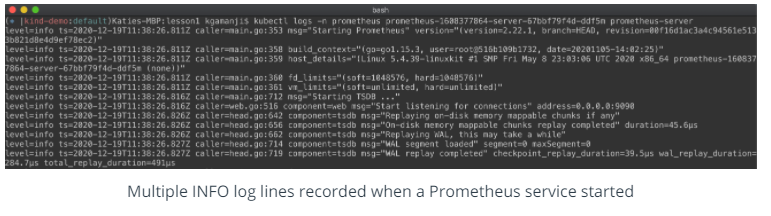
Log aggregation provides valuable insights into what operations a service is performing at a point in time. It is the nucleus of any troubleshooting and debugging process. For example, it is essential to record if a user logged in successfully into a service, or encountered an error while performing a payment.

Usually, the logs are collected from STDOUT (standard out) and STDERR (standard error) through a passive logging mechanism. This means that any output or errors from the application are sent to the shell. Subsequently, these are collected by a logging tool, such as Fluentd or Splunk, and stored in backend storage. However, the application can send the logs directly to the backend storage. In this case, an active logging technique is used, as the log transmission is handled directly by the application, without a logging tool required.

There are multiple logging levels that can be attributed to an operation. Some of the most widely used are:

* **DEBUG** - record fine-grained events of application processes
* **INFO** - provide coarse-grained information about an operation
* **WARN** - records a potential issue with the service
* **ERROR** - notifies an error has been encountered, however, the application is still running
* **FATAL** - represents a critical situation, when the application is not operational

As well, it is common practice to associate each log line with a **timestamp**, that will exactly record when an operation was invoked.

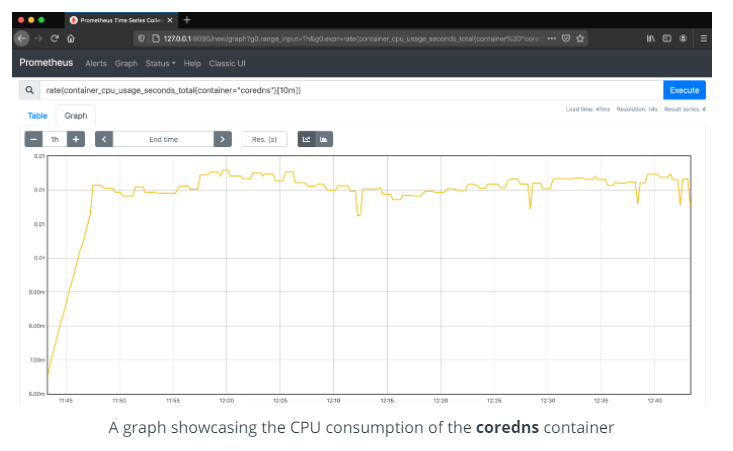


**Tracing**

Tracing is capable of creating a full picture of how different services are invoked to fulfill a single request. Usually, tracing is integrated through a library at the application layer, where the developer can record when a particular service is invoked. These records for individual services are defined as spans. A collection of spans define a trace that recreates the entire lifecycle of a request.

**Resource Consumption**

Resource consumption encapsulates the resources an application requires to be fully operational. This usually refers to the amount of CPU and memory that is consumed by an application during its execution. Additionally, it is beneficial to benchmark the network throughput, or how many requests can an application handle concurrently. Having awareness of resource boundaries is essential to ensure that the application is up and running 24/7.



**Further reading**

* [Health Checks](https://microservices.io/patterns/observability/health-check-api.html) - explore the core reasons to introduce health checks and implementations examples
* [Prometehus Best Practices on Metrics Naming](https://prometheus.io/docs/instrumenting/writing_exporters/#metrics) - explore how to name, label, and define the type of metrics
* [Application Logging Best Practices](https://logz.io/blog/logging-best-practices/) - read more on how to define what logs should be collected by an application
* [Logging Levels](https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm) - explore possible logging levels and when they should be enabled
* [Enabling Distributed Tracing for Microservices With Jaeger in Kubernetes](https://containerjournal.com/topics/container-ecosystems/enabling-distributed-tracing-for-microservices-with-jaeger-in-kubernetes/) - learn what tools can be used to implement tracing in a Kubernetes cluster