**Dapr Observability**

Modern distributed systems exhibit a high level of complexity. The initial phase involves creating small, independently deployable services that are loosely connected. These services traverse both process and server boundaries, interacting with various infrastructure backing services such as databases, message brokers, and key vaults.

Ultimately, these diverse components come together to constitute a cohesive application. Outdated monitoring approaches from the past fall short. Instead, the system necessitates comprehensive observability from end to end. Modern observability practices offer continuous visibility and valuable insights into the application's health at all times. They enable you to infer the internal state by observing the output. And I need to mention that it is crucial to implement this at the beginning of the project

The system information utilized for achieving observability is commonly known as telemetry, and it can be categorized into four broad groups:

* Distributed tracing offers visibility into the communication among services engaged in distributed business transactions.
* Metrics offer insights into the performance of a service and its resource utilization.
* Logging provides visibility into the execution of code and the occurrence of errors.
* Health endpoints offer insights into the availability of a service.

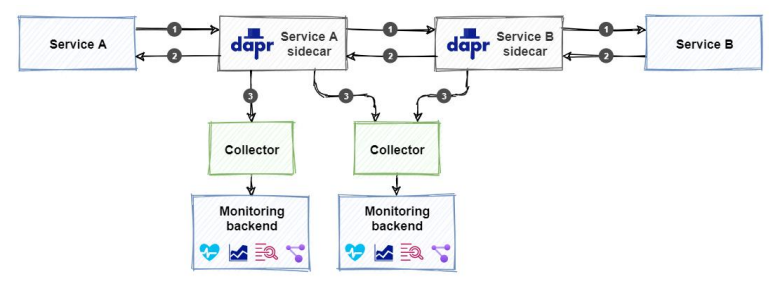
**The problem Dapr Observability tackles**

The observability component of Dapr decouples observability from the application. It automatically records the activity produced by Dapr sidecars and Dapr system services constituting the Dapr control plane. This component harmonizes traffic associated with a specific operation that spans across diverse services.

Additionally, it provides access to performance metrics, resource consumption details, and overall system health. Telemetry data is disseminated in open-standard formats, facilitating integration with the monitor Telemetry data is published in open-standard formats, facilitating integration with the monitoring backend services. This allows for the visualization, querying, and analysis of the information in your chosen monitoring environment. By abstracting the underlying infrastructure details, Dapr shields the application from the complexity of observability implementation. Dapr enables developers to prioritize the creation of business logic over dealing with the intricacies of observability plumbing. The configuration of observability is handled at the Dapr system level, ensuring uniformity across services, even when developed by different teams and built with varied technology stacks.

**Dapr operational mechanism**

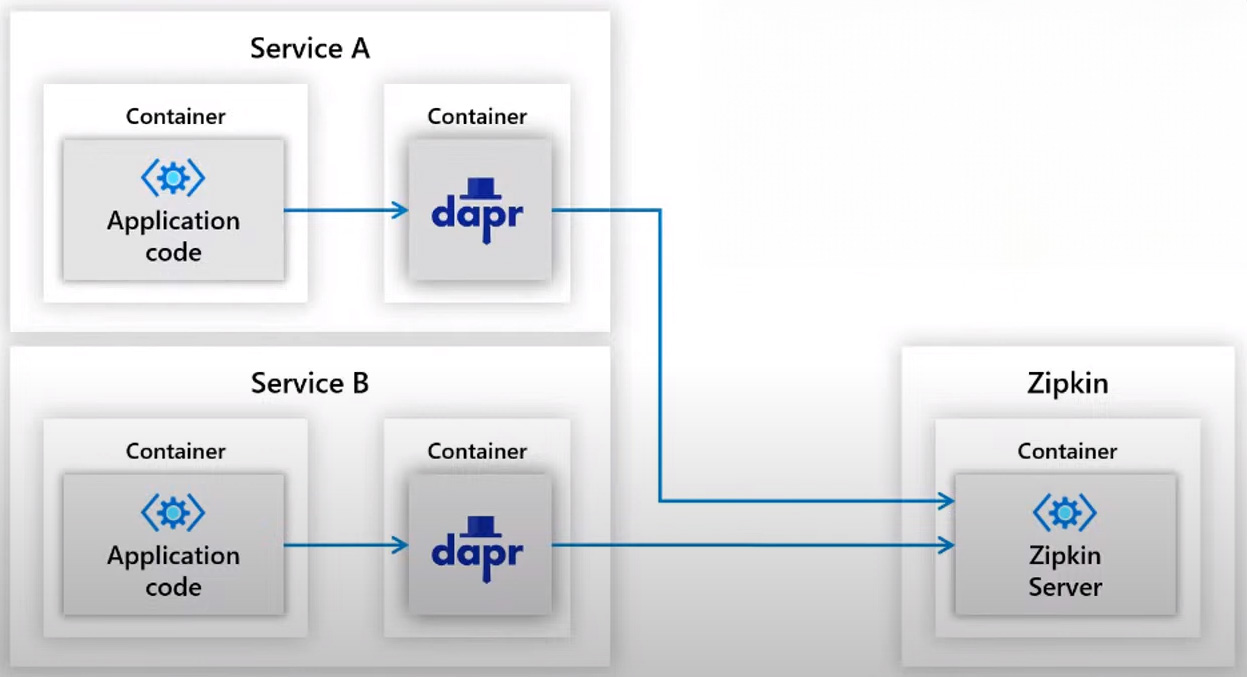
Dapr’s sidecar architecture enables built-in observability features. As various services interact, Dapr sidecars intercept the communication flow, extracting essential tracing, metrics, and logging data. Telemetry is published in an open standards format. By default, Dapr supports OpenTelemetry and Zipkin. Dapr supplies collectors capable of dispatching telemetry to diverse backend monitoring tools. These tools facilitate the analysis and querying of Dapr telemetry data.



It's crucial to note that incorporating observability differs from configuring other Dapr building blocks, such as pub/sub or state management. Instead of referencing a specific building block, you integrate a collector and a monitoring backend. You can use Dapr either in a self-hosted or on Kubernetes. In this article we will focus on the self-hosted mode.

**Distributed Tracing**

Distributed tracing offers visibility into the communication between services within a distributed application. Analyzing the logs containing exchanged request and response messages is crucial for troubleshooting issues. The challenge lies in connecting messages associated with the same business transaction. Dapr utilizes the W3C Trace Context to link related messages. It incorporates this context information into both requests and responses, forming a distinct operation. Tracing works by injecting a correlation token in requests and responses, automatically handled by the Dapr sidecar when configured and has multiple monitoring backends available for ingesting and visualizing this information as Zipkin, Jaeger, New Relic.



Overview of telemetry pushing to Zipkin Server

**Configure Dapr Tracing**

Below you can see the Dapr config.yaml example for tracing. In yaml file tracing enabled and endpoint is Zipkin we can use another monitoring service for it. Zipkin automatically installed when you install Dapr and tracing is enabled in the default configuration file. This file located in $HOME/.dapr/.config.yaml or %USERPROFILE%\.dapr\.yaml on Windows.

apiVersion: dapr.io/v1alpha1

kind: Configuration

metada:

name: daprConfig

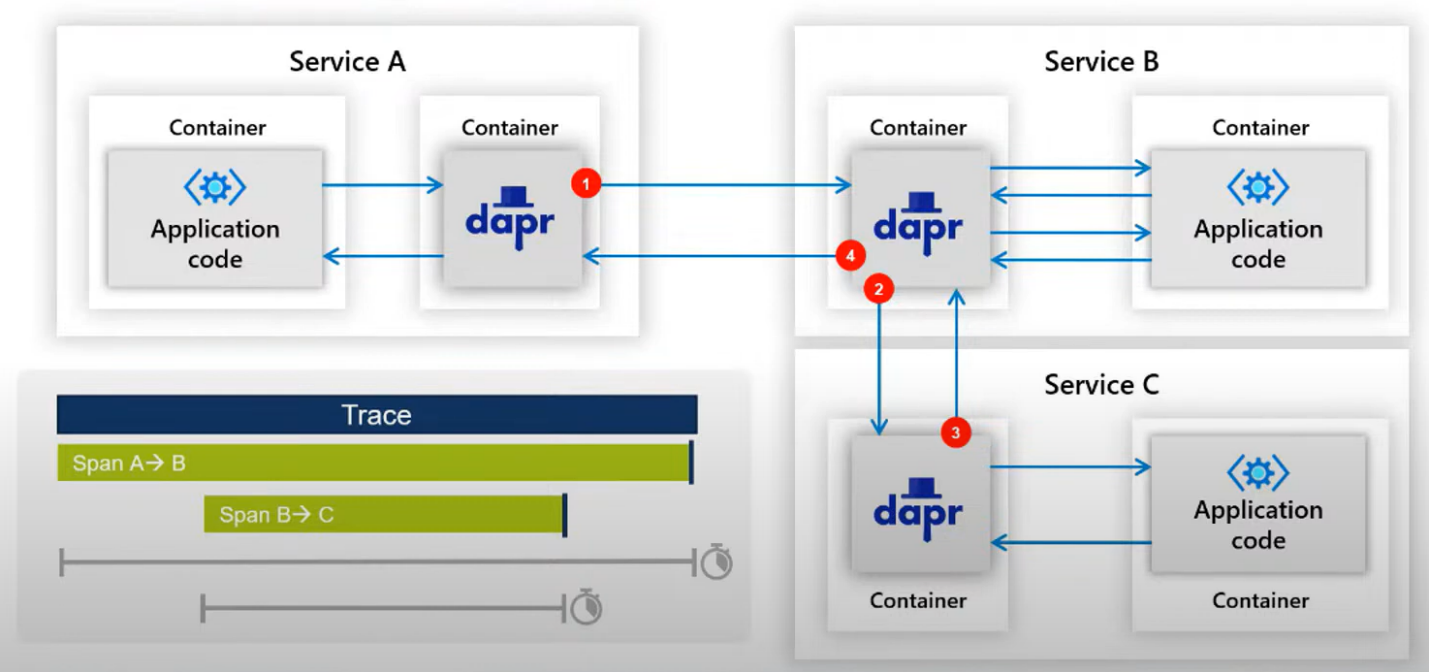
spec:

tracing:

samplingRate: "1"

zipkin:

endpointAddress: http:/localhost:9411/api/v2/spans

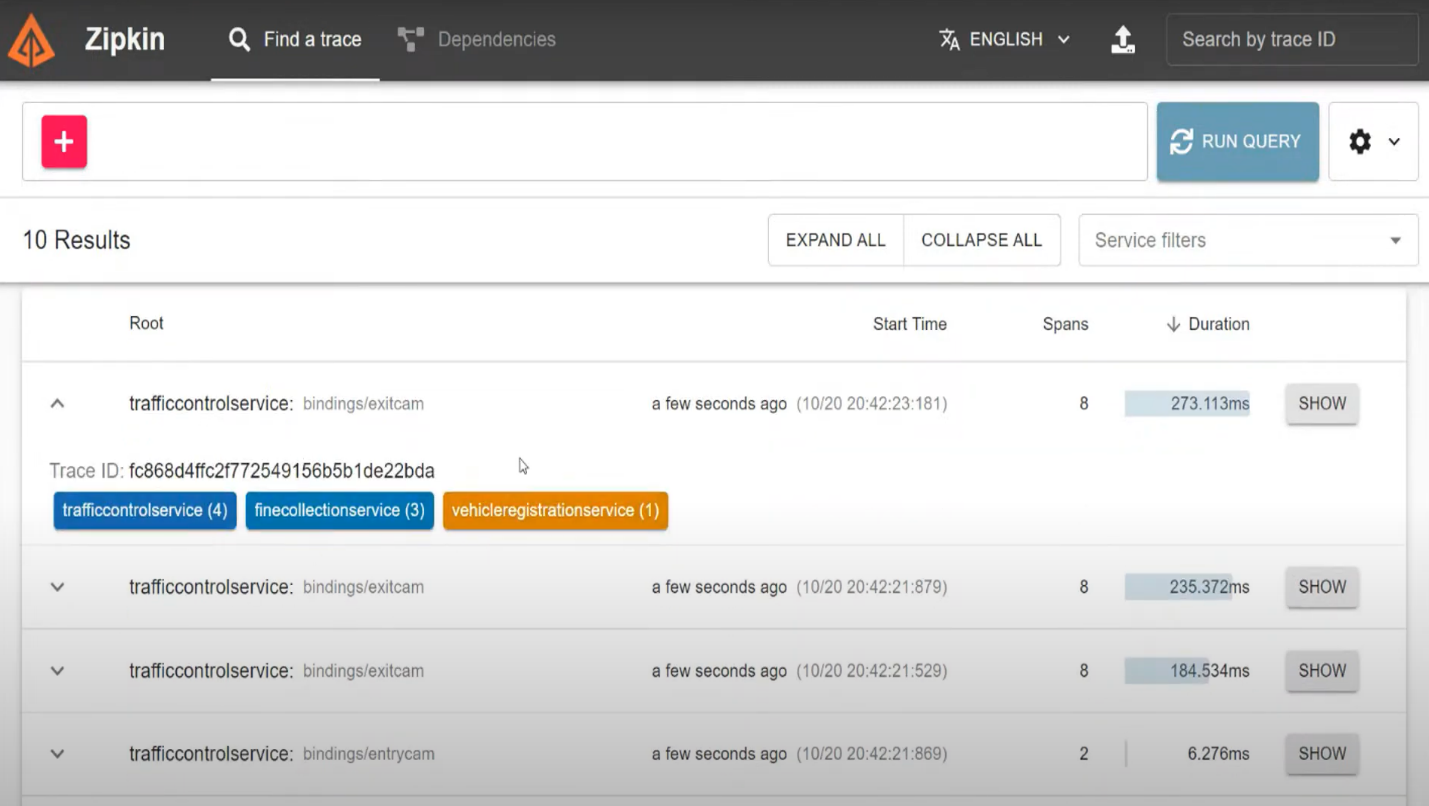


1. Service A calls Service B and then Dapr sidecar is forwarding this call. It will pushed this to telemetry to Zipkin Server. Trace and span will be started.
2. Service B calls Service C through the Dapr sidecar when this call is done telemetry will be sent to Zipkin Server and there will be span A->B progress and also new span will be recorded B->C
3. When Service C responses and after that Service B responses and telemetry will be sent to Zipkin Server and Trace will be done end to end.

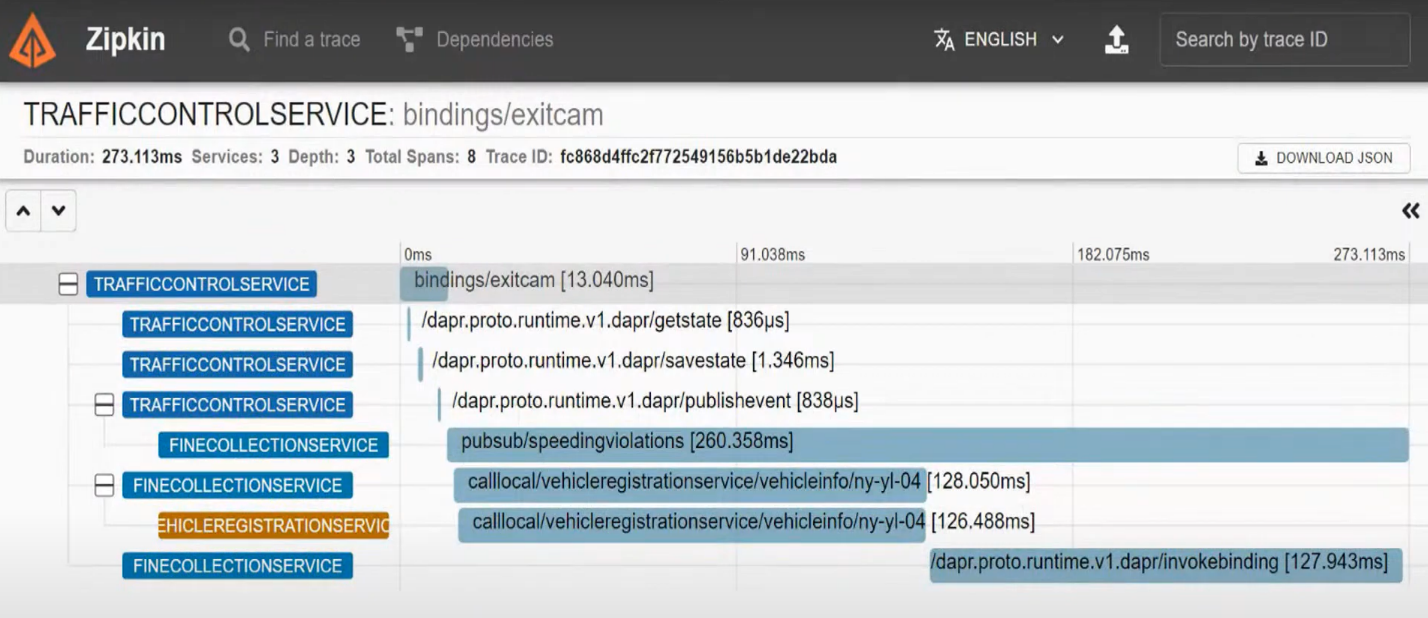
Dapr sidecar also records timestamps of these spans. So it provides us to how long does these calls take.

Dapr created a unique trace context and inject it into the request-response chain. The trace context is referred to as a correlation token in microservices terminology.

I use dapr-traffic-control repository as a reference microservices project to visualize tracing. In the repository you can find information about how to run the relevant project. Repository link can be found in references.



Clicking run query button will show all the ingested traces.

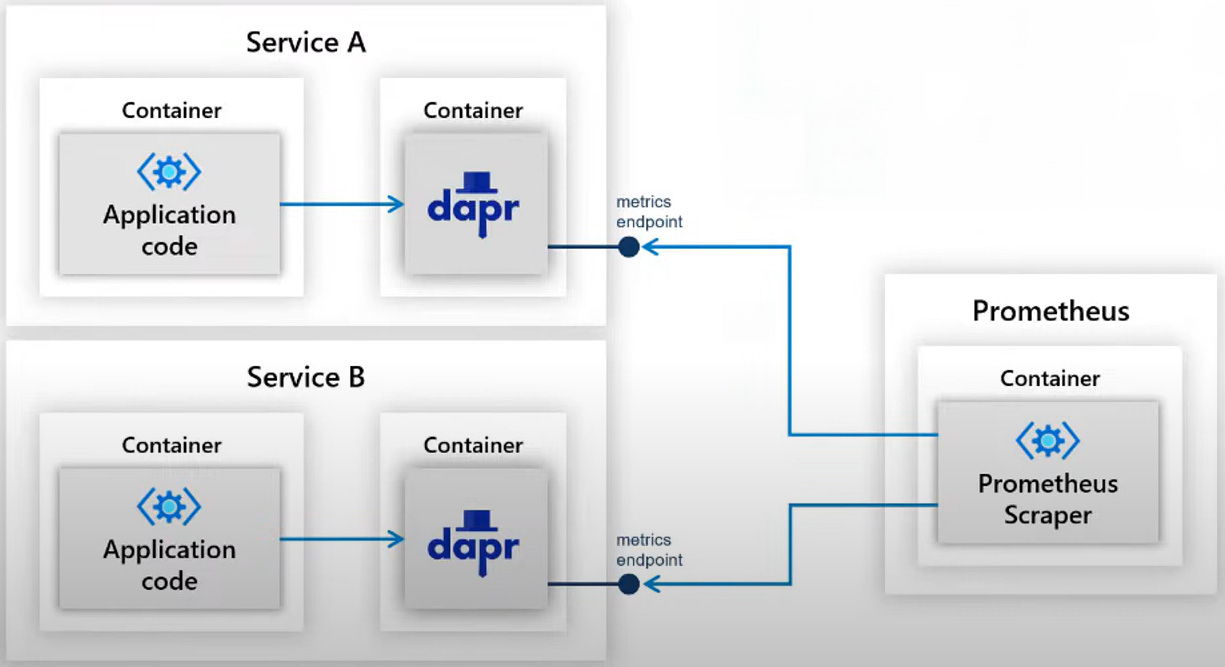


Clicking the show button of any of these traces will show the detailed information about that trace. You can see the entire trace with spans that have multiple services and multiple communication mechanisms.

1. First of all we see the binding that coming into the Traffic Control Service and its recorded by Dapr and push to Zipkin.
2. Also we can see the state management communication and publish&subscribe communication which are recorded in Zipkin.

**Metrics**

Metrics offer valuable understanding of both performance and resource utilization. Dapr uses Prometheus as a metric standard. Dapr sidecars and system services expose a metrics endpoint on port 9090. The Prometheus metrics scraper captures performance and resource consumption metrics from each endpoint and publishes the information to the monitoring backend. There are different levels of metrics such as Dapr system metrics, Dapr sidecar metrics, Dapr actors. All monitoring backends compatible with Promethues can be used for ingesting and visualizing this information. (Promethues, Grafana, Azure Monitor, New Relic)



**Configure Dapr Metrics**

Below you can see the Dapr config.yaml example for metrics. This file located in $HOME/.dapr/.config.yaml or %USERPROFILE%\.dapr\.yaml on Windows.

apiVersion: dapr.io/v1alpha1

kind: Configuration

metada:

name: daprConfig

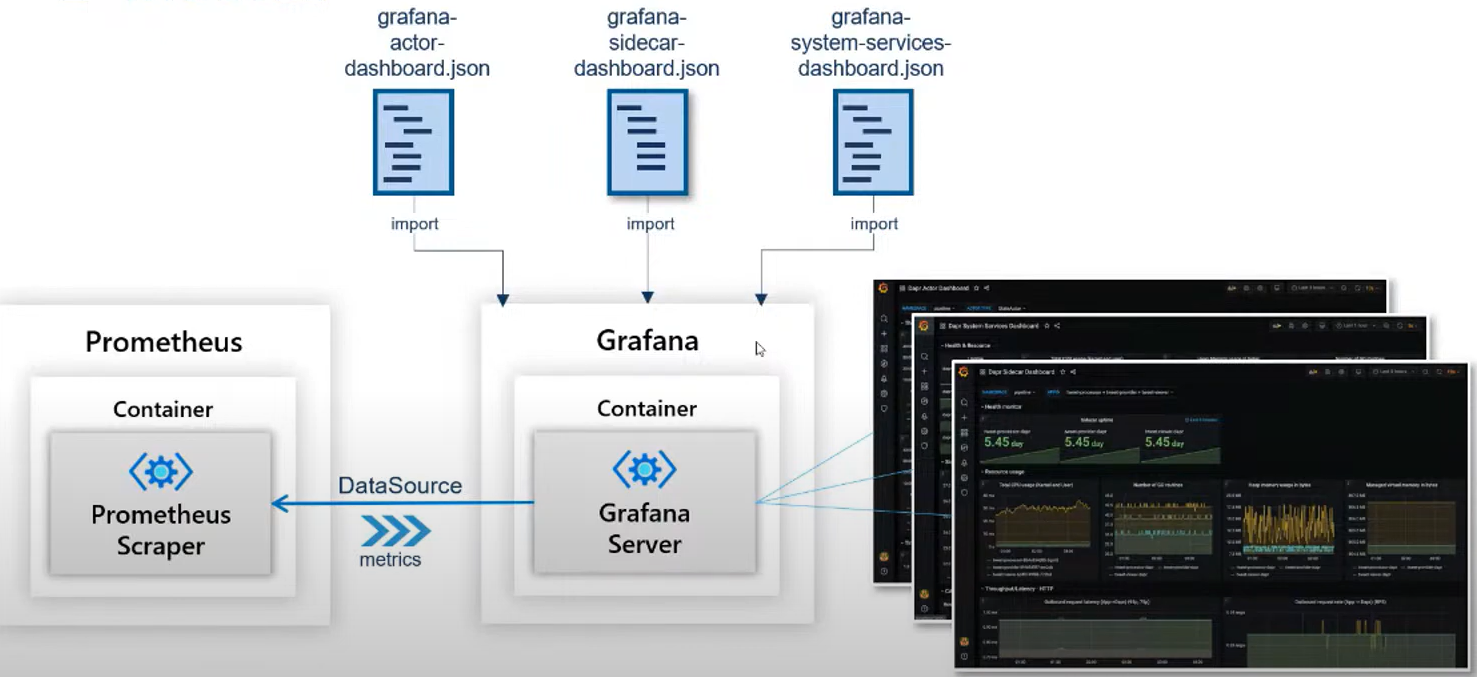
spec:

metric:

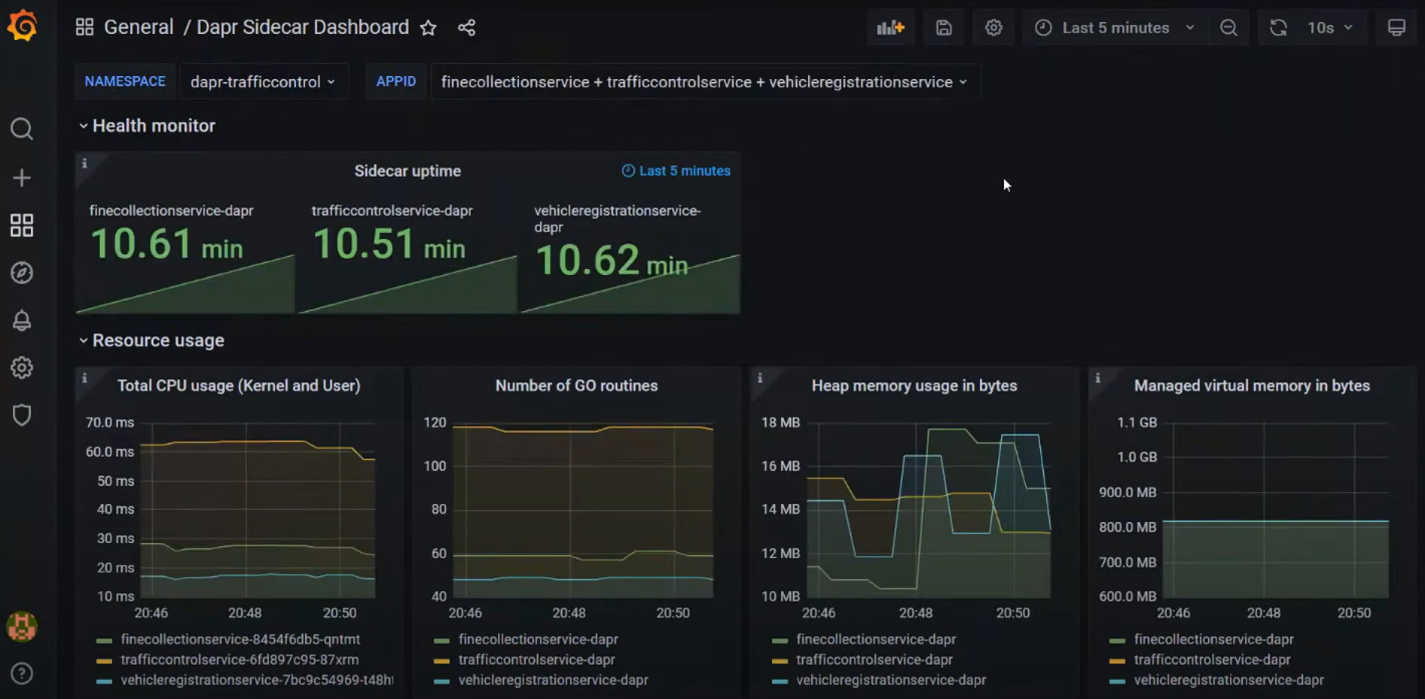
enabled: true

**Visualize Dapr Metrics**

A popular visualization tool for analyzing metrics is Grafana. With Grafana you can create dashboards from available metrics. In Grafana we can specify the Promotheus Server as datasource. We need to import our dashboard configuration to Grafana. You can provide these configurations from Dapr Github Repository.



After importing dashboard configuration json file to Grafana. You can access a dashboard like below.



**Logging**

Logging offers visibility into the real-time activities of a service. When an application is in operation, Dapr inherently generates log entries from Dapr sidecars and Dapr system services. Yet, log entries introduced in your application code are not automatically incorporated. To include logging from your application code, it's possible to integrate a specific SDK, such as the OpenTelemetry SDK for .NET.

**Log Entry Structure**



**Health Status**

The availability of a service can be understood by examining its health status. Eacher Dapr sidecar has a health API that the hosting environment can utilize to assess the sidecar’s health.

The API has one operation: GET <http://localhost:3500/v1.0/healthz>

* 204 means the sidecar is healthy.
* 500 means the sider is not healthy.

In self-hosted mode, the health API is not triggered automatically. However, you have the option to manually invoke the API from your application code or using a health monitoring tool. When running in Kubernetes, the dapr sidecar-injector automatically configures Kubernetes to use the health API for executing liveness probes and readiness probes.

References

* Dapr for .NET Developers, eBook
* https://docs.dapr.io/
* <https://github.com/EdwinVW/dapr-traffic-control/tree/main>