Prometheus Monitoring Mixins

Using jsonnet to package together dashboards, alerts and exporters.

Status: Draft
Tom Wilkie, *Grafana Labs*
Frederic Branczyk, *Red Hat*

In this design doc we present a technique for packaging and deploying “Monitoring Mixins” - extensible and customisable combinations of dashboards, alert definitions and exporters.

Problem

Prometheus offers powerful open source monitoring and alerting - but that comes with higher degrees of freedom, making pre-configured monitoring configurations hard to build. Simultaneously, it has become accepted wisdom that the developers of a given software package are best placed to operate said software, or at least construct the basic monitoring configuration.

This work aims to build on Julius Volz’ document “Prometheus Alerting and Dashboard Example Bundles“ and subsequent PR “Add initial node-exporter example bundle”. In particular, we support the hypothesis that for Prometheus to gain increased traction we will need to appeal to non-monitoring-experts, and allow for a relatively seamless pre-configured monitoring experience. Where we disagree is around standardisation: we do not want to prescribe a given label schema, example deployment or topology. That being said, a lot of the challenges surfaced in that doc are shared here.

Aims

This solution aims to define a minimal standard for how to package together Prometheus alerts, Prometheus recording rules and Grafana dashboards in a way that is:

Easy to install and use, platform agnostic. The users of these packages are unlikely to be monitoring experts. These packages must be easily installable with a few commands. And they must be general enough to work in all the environments where Prometheus can work: we’re not just trying to build for Kubernetes here. That being said, the experience will be first class on Kubernetes.

Hosted alongside the programs which expose Prometheus metrics. More often than not, the best people to build the alerting rules and dashboards for a given application are the authors of that application. And if that is not the case, then at least users of a given application will look

to its source for monitoring best practices. We aim to provide a packaging method which allows the repo hosting the application source to also host the applications monitoring package; for them to be versioned along side the application. For example, we envisage the monitoring mixin for Etcd to live in the etcd repo and the monitoring package for Hashicorp's Consul to live in the consul_exporter repo.

We want the ability to iterate and collaborate on packages. A challenge with the existing published dashboards and alerts is that they are static: the only way to use them is to copy them into your codebase, edit them to make them fit with your deployment. This makes it hard for users to contribute changes back to the original author; it makes it impossible to download new improved versions and stay up to date with improvements. We want these packages to be constantly evolving; we want to encourage drive-by commits.

Packages should be reusable, configurable and extensible. Users should be able to configure the packages to fit their deployments and labels schema without modifying the packages. Users should be able to extend the packages with extra dashboard panels and extra alerts, without having to copy, paste and modify them. The packages must be configurable so that they support the many different label schemes used today by different organisations.

Proposal

Monitoring Mixins. A monitoring mixin is a package of configuration containing Prometheus alerts, Prometheus recording rules and Grafana dashboards. Mixins will be maintained in version controlled repos (eg git) as a set of files. Versioning of mixins will be provided by the version control system; mixins themselves should not contain multiple versions.

Mixins are intended just for the combination of Prometheus and Grafana, and not other monitoring or visualisation systems. Mixins are intended to be opinionated about the choice of monitoring technology.

Mixins should not however be opinionated about how this configuration should be deployed; they should not contain manifests for deploying Prometheus and Grafana on Kubernetes, for instance. Multiple, separate projects can and should exist to help deploy mixins; we will provide example of how to do this on Kubernetes, and a tool for integrating with traditional config management systems.

Jsonnet. We propose the use of jsonnet, a configuration language from Google, as the basis of our monitoring mixins. Jsonnet has some popularity in this space, as it is used in the ksonnet project for achieving similar goals for Kubernetes.

Jsonnet offers the ability to parameterise configuration, allow for basic customisation. Furthermore, in Jsonnet one can reference another part of the data structure, reducing

repetition. For example, with jsonnet one can specify a default job name, and then have all the alerts use that:

```
{
  _config+:: {
    kubeStateMetricsSelector: 'job="default/kube-state-metrics"',

    allowedNotReadyPods: 0,
  },

  groups+: [
    {
      name: "kubernetes",
      rules: [
        {
          alert: "KubePodNotReady",
          expr: |||
            sum by (namespace, pod) (
              kube_pod_status_phase{%(kubeStateMetricsSelector)s, phase!~"Running|Succeeded"}
            ) > $(allowedNotReadyPods)s
          ||| % $._config,
          "for": "1h",
          labels: {
            severity: "critical",
          },
          annotations: {
            message: "{{ $labels.namespace }}/{{ $labels.pod }} is not ready.",
          },
        },
      ],
    },
  ],
}
```

Configuration. We'd like to suggest some standardisation of how configuration is supplied to mixins. A top level `_config` dictionary should be provided, containing various parameters for substitution into alerts a dashboards. In the above example, this is used to specify the selector for the kube-state-metrics pod, and the threshold for the alert.

Extension. One of jsonnet's basic operations is to "merge" data structures - this also allow you to extend existing configurations. For example, given an existing dashboard:

```
local g = import "klumps/lib/grafana.libsonnet";

{
  dashboards+:: {
    "foo.json": g.dashboard("Foo")
      .addRow(
        g.row("Foo")
        .addPanel(
          g.panel("Bar") +
          g.queryPanel('irate(foor_bar_total[1m])', 'Foo Bar')
        )
      )
```

```
  },
}
```

It is relatively easy to import it and add extra rows:

```
local g = import "foo.libsonnet";

{
  dashboards+:: {
    "foo.json"+:
      super.addRow(
        g.row("A new row")
        .addPanel(
          g.panel("A new panel") +
          g.queryPanel('irate(new_total[1m])', 'New')
        )
      )
  },
}
```

These abilities offered by jsonnet are key to being able to separate out “upstream” alerts and dashboards from customizations, and keep upstream in sync with the source of the mixin.

Higher Order Abstractions. jsonnet is a functional programming language, and as such allows you to build higher order abstractions over your configuration. For example, you can build functions to generate recording rules for a set of percentiles and labels aggregations, given a histogram:

```
local histogramRules(metric, labels) =
  local vars = {
    metric: metric,
    labels_underscore: std.join("_", labels),
    labels_comma: std.join(", ", labels),
  };
  [
    {
      record: "%(labels_underscore)s:%(metric)s:99quantile" % vars,
      expr: "histogram_quantile(0.99, sum(rate(%(metric)s_bucket[5m])) by (le,
%(labels_comma)s))" % vars,
    },
    {
      record: "%(labels_underscore)s:%(metric)s:50quantile" % vars,
      expr: "histogram_quantile(0.50, sum(rate(%(metric)s_bucket[5m])) by (le,
%(labels_comma)s))" % vars,
    },
    {
      record: "%(labels_underscore)s:%(metric)s:avg" % vars,
      expr: "sum(rate(%(metric)s_sum[5m])) by (%(labels_comma)s) /
sum(rate(%(metric)s_count[5m])) by (%(labels_comma)s)" % vars,
    },
  ];

{
```

```
    groups+: [{
      name: "frontend_rules",
      rules:
        histogramRules("frontend_request_duration_seconds", ["job"]) +
        histogramRules("frontend_request_duration_seconds", ["job", "route"]),
    }],
}
```

Other potential examples include functions to generate alerts at different thresholds, omitting multiple alerts, warning and critical.

Grafonnet. An emerging pattern in the jsonnet ecosystem is the existence of libraries of helper functions to generate objects for a given system. For example, ksonnet is a library to generating objects for the Kubernetes object model. Grafonnet is a library for generating Grafana Dashboards using jsonnet. We envisage a series of libraries, such as Grafonnet, to help people build mixins. As such, any system for installing mixins needs to deal with transient dependencies.

Package Management. The current proof of concepts for mixins (see below) use the new package manager jsonnet-bundler enabling the following workflow:

```
$ jb install kausal github.com/kausalco/public/consul-mixin
```

This downloads a copy of the mixin into `vendor/consul-mixin` and allows users to include the mixin in their ksonnet config like so:

```
local prometheus = import "prometheus-ksonnet/prometheus-ksonnet.libsonnet";
local consul_mixin = import "consul-mixin/mixin.libsonnet";

prometheus + consul_mixin {
  _config+:: {
    namespace: "default",
  },
}
```

This example also uses the prometheus-ksonnet package from Kausal, which understands the structure of the mixins and manifests alerting rules, recording rules and dashboards as config maps in Kubernetes, mounted into the Kubernetes pods in the correct place.

However, we think this is a wider problem than just monitoring mixins, and are exploring designs for a generic jsonnet package manager in a separate design doc.

Proposed Schema. To allow multiple tools to utilise mixins, we must agree on some common naming. The proposal is that a mixin is a single dictionary containing three keys:

- `grafanaDashboards` A dictionary of dashboard file name (foo.json) to dashboard json.

- `prometheusAlerts` A list of Prometheus alert groups.
- `prometheusRules` A list of Prometheus alert groups.

Each of these values will be expressed as jsonnet objects - not strings. It is the responsibility of the tool consuming the mixin to render these out as JSON or YAML. Jsonnet scripts to do this for you will be provided.

```
{
  grafanaDashboards+:: {
    "dashboard-name.json": {...},
  },
  prometheusAlerts+:: [...],
  prometheusRules+:: [...],
}
```

Consuming a mixin.

- TODO examples of how we expect people to install, customise and extend mixins.
- TODO Ability to manifest out jsonnet configuration in a variety of formats - YAML, JSON, INI etc
- TODO show how it works with ksonnet but also with something like puppet..

Examples & Proof of Concepts

We will probably put the specification and list of known mixins in a repo somewhere, as a readme. For now, these are the known mixins and related projects:

Application	**Mixin**	**Author**
CoreOS Etcd	etcd-mixin	Grapeshot / Tom Wilkie
Cassandra	TBD	Grafana Labs
Hashicorp Consul	consul-mixin	Kausal
Hashicorp Vault	vault_exporter	Grapeshot / Tom Wilkie
Kubernetes	kubernetes-mixin	Tom Wilkie & Frederic Branczyk
~~Kubernetes~~	~~KLUMPS~~	~~Kausal~~
Kubernetes	kubernetes-grafana	Frederic Branczyk
Kubernetes	kube-prometheus	Frederic Branczyk
Prometheus	prometheus-ksonnet	Kausal
Prometheus Node Exporter	TBD	

Open Questions

- Some systems require exporters; can / should these packaged as part of the mixin? Hard to do generally, easy to do for kubernetes with ksonnet.
- On the exporter topic, some systems need stats_exporter mappings to be consistent with alerts and dashboards. Even if we can include statds_exporter in the mixin, can we include the mappings?
- A lot of questions from Julius' design are still open: how to deal with different aggregation windows, what labels to use on alerts etc.