Metrics

Overview

We want to unify our reporting infrastructure so that we have a centralized way to report on metrics related the health and behavior of the helix services. We'll have a single place for defining all the interesting information about a metric (it's names, which services it applies to, levels it should be alerted at…), which we'll generate all the pieces we need. * Strongly typed wrappers to send metrics * Automatically deployed alerting rules * Visualizations These metrics will either be sent by the services themselves, or an external monitoring service.

Unified File

We will have a centralized file to define all our metrics, something similar to

```yaml
services:
  - missionControl
  - helixApi
instances:
  - helix
  - missionControl
  - helixClient
metrics:
  -
    name: DataMigrationQueueDepth
    instance: helix
    servicesAffected:
      - missionControl
      - helixApi
    warning: >100
    error: >1000
    alertIfMissing: true
```

This will define the services we monitor, the application insights instances that we are reporting to/from, and then a list of all metrics that we're report and alert.

Code Generation

From the unified file, we'll run some pre-processing code to generate C# (and potentially other language) wrappers for reporting,

```csharp
public class HelixMetric
{
  public void Track(double value);
}
```

```
public class HelixReportingProvider
{
  public HelixMetric DataMigrationQueueDepth { get; }
}

...

private HelixReportingProvider _reporting; // DI this

public void SomewhereElse()
{
  _reporting.DataMigrationQueueDepth.Track(900);
}
```

We'll handle pre-aggregation based on recommendations here. The metrics/events will be reporting to AI as customMetrics and customEvents If we need wrappers for other languages, we can work on that.

The wrappers will be generated at build time, so there won't be a need to check them in, hopefully.

Monitoring Service

Most monitoring can be handled by the services themselves, for example, when pulling an item from an Azure queue, it's trivial to report on the age of the item at that point (since it's part of the pulled item).

However, sometimes this isn't true. For example, reporting on the queue *depth* isn't something that's easily handled at pull time, and it shouldn't be done by every pulling instance, since it would just cause noise (there is only one queue depth to report). These will be handled by a separate monitoring service that runs and monitors external factors, like queue depth, or reachability of storage accounts, or such things.

Reporting

We're going to use Power BI to start and see if that's a good fit for our alerting. Hopefully we can make it data driven as well, so that it can pull from our source file with some transformation.

If Power BI doesn't work out, we'll use custom graphing in Mission Control, pulling from AI

Outage Service

Knowing a metric is off is great for our ability to run our service, but an important part of a service is the ability to report out to users about downtime and outages.

To this end, we need to create a fairly simple outage reporting service. It doesn't need to be much more than a table that lists services, any outages, and any notes that we have made about it's status.

There should be an API listening here that AI alerts can report to. When that happens, we should open some sort of ticket (probably a github issue) so that it can be tracked/resolved.

This won't consist of more than a fairly simple Azure Table, and a handful of webpages (overall status, report new outage, resolve existing outage).

Alerting

Application Insights already has fairly good alerting based on custom metrics, which we will take advantage of. Using the single file, a resource template will be created, as shown here. It will both email the outage to an alias (maybe dnceng) and notify the outage service so it can open an issue and possible mark an outage.

All of these alerts will alert to the same email address and also report to an alerting service.

What to Measure

In short: whatever it's feasible to measure. Having a metric reporting should be very low cost to the service, as long as it's not in a very tight inner loop, so over-reporting should be prefered to under-reporting.

Some examples of things that we should measure … * … in any sort of producer/consumer model … * … the average production and consumption rates * … the current depth of the backlog * … the average delay between production and consumption * … in any user exposed service … * … availablity * … average response times for a fixed, known piece of work * … in a destributed service … * … heartbeats inside the code that's doing work * … of any external service we depend on * … availability query (can we contact the service with the credentials we expect)

Was this helpful?